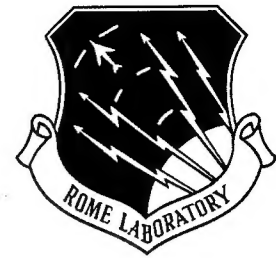


RL-TR-97-44
Final Technical Report
July 1997



SUPPORTING MULTIUSER ACCESS TO LARGE-SCALE PERSISTENT KNOWLEDGE BASES

SRI International

Sponsored by
Advanced Research Projects Agency
ARPA Order No. 8964

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

19970922 088

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

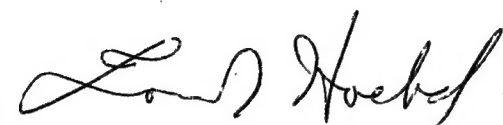
[DTIC QUALITY INSPECTED 3]

Rome Laboratory
Air Force Materiel Command
Rome, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

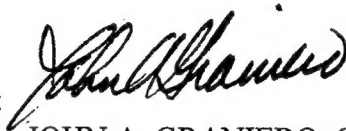
RL-TR-97-44 has been reviewed and is approved for publication.

APPROVED:



LOUIS J. HOEBEL
Project Engineer

FOR THE COMMANDER:



JOHN A. GRANIERO, Chief Scientist
Command, Control, & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL/C3CA, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

SUPPORTING MULTIUSER ACCESS TO LARGE-SCALE
PERSISTENT KNOWLEDGE BASES

Contractor: SRI International
Contract Number: F30602-92-C-0115
Effective Date of Contract: 12 August 1992
Contract Expiration Date: 30 June 1996
Program Code Number: 2D30
Short Title of Work: Supporting Multiuser Access to Large-
Scale Persistent Knowledge Bases
Period of Work Covered: Aug 92 - Jun 96

Principal Investigator: Peter D. Karp
Phone: (415) 859-6323
RL Project Engineer: Louis J. Hoebel
Phone: (315) 330-3655

Approved for public release; distribution unlimited.

This research was supported by the Advanced Research Projects
Agency of the Department of Defense and was monitored by
Louis J. Hoebel, RL/C3CA, 525 Brooks Road, Rome, NY.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE July 1997		3. REPORT TYPE AND DATES COVERED Final Aug 92 - Jun 96
4. TITLE AND SUBTITLE SUPPORTING MULTIUSER ACCESS TO LARGE-SCALE PERSISTENT KNOWLEDGE BASES			5. FUNDING NUMBERS C - F30602-92-C-0115 PE - 61101E PR - H767 TA - 00 WU - 02	
6. AUTHOR(S) Peter D. Karp				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) SRI International 333 Ravenswood Ave Menlo Park CA 94025			8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Advanced Research Projects Agency 3701 North Fairfax Drive Arlington VA 22203-1714			10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-97-44	
11. SUPPLEMENTARY NOTES Rome Laboratory Project Engineer: Louis J. Hoebel/C3CA/(315) 330-3655				
12a. DISTRIBUTION AVAILABILITY STATEMENT Approved for public release; distribution unlimited			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) This project has investigated methods for extending the capabilities of the loom frame representation system (FRS) to support the development of large-scale knowledge bases by multiple, distributed users.				
14. SUBJECT TERMS Persistence, concurrent access			15. NUMBER OF PAGES 64	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

Abstract

This project has investigated methods for extending the capabilities of the LOOM frame representation system (FRS) to support the development of large-scale knowledge bases (KBs) by multiple, distributed users. SRI has explored alternative methods for coupling LOOM with a DBMS. Those methods were implemented at SRI, evaluated experimentally, and published in the computer-science literature [KPG94, KP95]. SRI has also developed and implemented a novel optimistic concurrency control technique for controlling KB updates by multiple users. The implementations were tested in the context of two domains: the SOCAP military-operations planning KB, and the EcoCyc biochemical-pathways KB.

The organization of this final report is as follows. Sections 2-4 discuss the motivations and design requirements for this work, and briefly present the architecture we have designed to satisfy those requirements.

Section 5 describes the Generic Frame Protocol (GFP), which was an unanticipated result of this work. GFP is a COMMON LISP interface that allows different FRSs to present a common programmer interface to FRS applications. GFP is a substrate for software reuse that has facilitated the reuse of the storage system with other FRSs. GFP also underlies both the Generic Knowledge Base Editor (see Section 6) that SRI has developed under a separate contract from Rome Laboratory, and the Ontology Editor developed at Stanford University.

Section 7 describes the LOOM storage system. SRI has implemented several alternative architectures for the storage system (based on a relational database management system (DBMS) and an object-oriented DBMS), and empirically evaluated the performance of those architectures. The two DBMSs exhibited similar performance, but the relational DBMS was easier to work with and therefore was chosen as the basis for future experiments. SRI designed a generic relational schema that can encode information from several LOOM KBs simultaneously, and that can accommodate any LOOM KB. SRI obtained a follow-on contract from DARPA to polish the storage system and distribute it to the LOOM community; that distribution will occur in the summer of 1996.

Section 8 describes our approach to coordinating KB updates from multiple simultaneous users. Our strategy is to allow users to freely update the KB in separate workspaces; when user wish to commit their changes to the public KB, the collaboration system searches for conflicts between changes made by the requesting user and other recent changes made by other users.

The following publications were prepared under this project:

- (in preparation) "A Collaborative Environment for Authoring Large Knowledge-Bases and Ontologies."
- Karp, P.D. and Paley, S. (1995) "Knowledge Representation in the Large," Proceedings of the 1995 International Joint Conference on Artificial Intelligence, Montreal, Canada. See WWW URL <ftp://ftp.ai.sri.com/pub/papers/karp-perkobj95.ps.Z>.
- Karp, P.D., Myers, K. and Gruber, T., (1995) "The Generic Frame Protocol," Proceedings of the 1995 International Joint Conference on Artificial Intelligence, Montreal,

Canada. See WWW URL <ftp://ftp.ai.sri.com/pub/papers/karp-gfp95.ps.Z>.

- Karp, P.D., Paley, S., and Greenberg, I., (1994) "A Storage System for Scalable Knowledge Representation," in *Proceedings of the Third International Conference on Information and Knowledge Management*, Gaithersburg, MD.

1 Introduction

Collaborative KB authoring environments allow multiple, geographically distributed users to collaborate in the development of large KBs. The first generation of FRs [Kar92] provided only single-user KB authoring environments whose engineering limitations constrained the size of the resulting KBs, and did not permit distributed KB access. This report describes a next-generation, reusable environment for collaborative KB authoring that consists of the following components:

- The Generic Frame Protocol, which provides infrastructure for software and knowledge reuse. It is a procedural interface to FRs that provides a common means of accessing and modifying frame KBs.
- The Storage Subsystem, which provides scalable storage of frame KBs in a commercial DBMS.
- The Collaboration Subsystem, which provides optimistic concurrency control over the KB updates made by multiple distributed KB authors.
- The GKB Editor, which provides KB browsing and editing services for large KBs.

This work makes a number of contributions to the field of knowledge representation. More specifically, our results advance the state-of-the-art in the tools for engineering knowledge-representation systems. We identify design requirements for collaborative KB authoring environments, present an architecture that satisfies those requirements and an implementation of that architecture. The architecture includes both the Storage Subsystem, for expanding the storage capabilities of FRs, and novel “optimistic” concurrency control techniques for coordinating KB updates made by multiple simultaneous users. We also present an empirical evaluation of our chosen architecture against a number of alternative architectures to illustrate their relative merits.

The GKB Editor completes this authoring environment by providing three different viewer and editor utilities for browsing and modifying large KBs. All of these tools have been used in the development of several real-world KBs including a planning ontology, and the EcoCyc biochemical-pathway KB, which contains 10,000 frames.

These results are made more significant because the implementations have been reused across several FRs. The Storage Subsystem has been used with both LOOM [Mac91] and THEO [MAC⁺89]; the GKB Editor has been used with LOOM, THEO, SIPE-2, and partially with ONTOLINGUA and CLASSIC. This reusability both supports the generality of our approach, and provides the most substantial example to date of the type of software reuse envisioned by the Knowledge Sharing Initiative [PFPS⁺92]. We discuss various obstacles to software reuse that we encountered, and the solutions that we devised.

2 Design Requirements

We derived the design requirements for KB authoring environments from our experiences with KBs in several domains. We present two scenarios of the use of these environments to illustrate the motivations behind these requirements.

Scenario 1 involves the distributed development of a planning ontology by multiple ARPA contractors who are distributed throughout the United States, as part of the ARPA /Rome Laboratory Planning Initiative. Contractors at roughly a dozen sites might wish to access the LOOM KB that implements a planning ontology, to browse the current state of the ontology, to add new class definitions interactively, and to edit existing definitions. The LOOM classifier runs during the editing process to infer relationships among new and existing classes. Users might also execute ontology translators to convert the planning ontology to other forms, such as a relational database.

Scenario 2 involves a biological KB that describes the biochemical reaction network within the *E. coli* cell. The KB models biological objects such as enzymes and bio-reactions. Experts Biologists from around the world will interact with the KB in several different ways. Some biologists wish to update the KB by editing that region of the KB that falls within their expertise. Other biologists require only read-only access to the KB: some will browse the KB using a graphical interface. Others will execute qualitative and quantitative simulation programs that model the metabolism of the cell. Other scientists will execute programs that redesign the biochemical network of *E. coli* for commercial purposes in bio-technology. Still other users will evaluate complex queries over the KB to answer scientific questions.

We have extracted the following design requirements from these scenarios.

- User sessions will involve several distinct patterns of KB operations:
 - Interactive browsing and editing sessions that access a small portion of a KB. Editing sessions may last hours or even days, and include changes to many different frames. The patterns of most KB modifications differ from database modifications that update a small number of isolated values.
 - Complex computations such as simulation, design, and expert systems that repeatedly access significant subsets of the KB.
 - Traditional database-like complex queries that access significant subsets of the KB, but that return relatively small amounts of data to the application.
- The next generation of KB sizes will range from 10^4 to 10^7 frames. Users may be geographically distributed, and may access the KB using long-distance Internet connections.

The schema-evolution capabilities of most FRSs should be retained. An attribute that distinguishes KBs from DBs is the large size of the schemas (number of class and relation definitions) that KBs tend to have. As KBs grow, we expect their schemas to grow also, as well as the need to modify class and relation definitions dynamically.

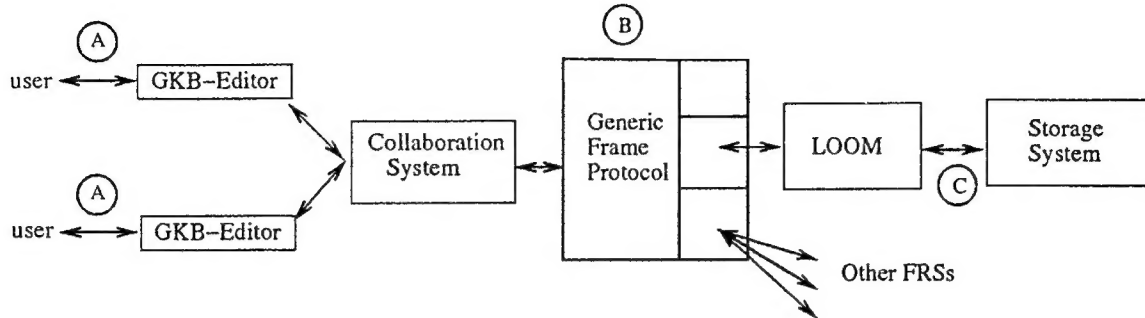


Figure 1: Proposed System Architecture

3 Architecture

Our system architecture is shown in Figure 1. Users employ the GKB Editor for interactive KB editing and browsing. It is reusable across multiple FRSs because all of its KB-access operations are implemented using the Generic Frame Protocol (GFP). GFP is a wrapping layer that allows multiple FRSs to present the same procedural interface to their applications. A GFP implementation exists for the LOOM FRS. Our architecture includes a storage subsystem that allows LOOM KBs to be stored within an ORACLE DB. Frames are incrementally retrieved on demand from the RDBMS into LOOM as the application executes. The storage subsystem also records what frames have been modified, and can incrementally save modified frames to the DBMS. Updates from multiple users are synchronized using a concurrency control method that detects conflicts at commit time between the updates made by a new user and other recent updates performed by other users. Our technique not only detects conflicts but also provides user assistance in resolving those conflicts. Thus, unlike traditional database techniques that keep an item locked for the whole duration of a transaction, we use locks only while the conflict-free updates are being deposited in the database.

The architecture of Figure 1 allows distributed operation because network links can be inserted between components at several places. We can insert a network link at (C) by transmitting Structured Query Language (SQL) calls to the ORACLE server over a network. We can insert a network link at (B) by creating a remote-procedure call version of GFP, in which every GFP call is sent over a network. We can insert a network link at (A) by allowing the X-window graphics of the GKB Editor to flow between an X client and X server.

A commercial DBMS can easily accommodate a KB of 10^6 frames. We prefer to use a commercial DBMS rather than implementing a new DBMS from scratch. We designed a "generic" DBMS schema that can accommodate any frame-based KB without forcing the user to design a new DBMS schema. The generic schema also facilitates evolution of the KB schema. The frame faulting approach allows compute-intensive AI applications such as planners, design programs, and simulators to operate in parallel on multiple client machines, on regions of the KB that are cached in the local memory of those client machines. AI applications tend to make a very large number of KB accesses, sometimes accessing the same frames repeatedly. If every access were transmitted over a long-distance Internet connection,

performance would be unacceptable.

4 Generic Frame Protocol

We first describe the goals of the Generic Frame Protocol (GFP). Next, we discuss the design principles of the GFP, decisions we had to make in the design process and how they were instrumental in achieving the general applicability of GFP. We then consider the implementation issues and empirical results evaluating the GFP and conclude the section with a discussion of related work.

4.1 Motivation/Objectives

The goal of GFP is to allow reuse of knowledge and software by mixing and matching knowledge and software components in a large AI system. By reuse of knowledge, we mean reusing a knowledge base or ontology developed in one formalism in another, for example importing an ontology developed in LOOM into Ontolingua. By reuse of software, we mean an easy porting of the software across other FRSs, for example, reusing a system such as GKB-Editor across multiple FRSs. GFP accomplishes its goals by defining a set of Common LISP generic functions that constitute a common application-program interface to FRSs. These functions constitute a wrapping layer for FRSs; that wrapping layer is implemented by creating FRS-specific methods that implement GFP operations.

4.2 Design Principles of GFP

Several design objectives were defined for GFP.

- **Simplicity:** The protocol should be simple and reasonably quick to implement for a particular FRS, even if this means sacrificing theoretical considerations or support for idiosyncrasies of that FRS.
- **Generality:** The protocol should apply to many FRSs, and support the most common FRS features.
- **No legislation:** The protocol should not require substantial changes to an FRS for which the protocol is implemented. That is, the protocol should not mandate the method of operation of an underlying FRS.
- **Performance:** Inserting the protocol between an application and an FRS should not introduce a significant performance cost.
- **Consistency:** The protocol should exhibit consistent behavior across implementations for different FRSs, that is, a given sequence of operations within the protocol should yield the same result over a range of FRSs.

- **Precision:** The specification of the protocol should be as precise and unambiguous as possible.
- **Language independence:** Ideally, GFP should be independent of programming language.

Satisfying these objectives simultaneously is impossible because many of them conflict. Different FRSs behave differently, and unless we mandate a minutely detailed behavioral model for KR systems (which no developers will subscribe to anyway), we cannot force these systems to behave the same. The GFP uses a knowledge model that encompasses many FRSs and is detailed enough to be useful in practice, but is not so detailed as to exclude every FRS from its model. Another example of conflicts among our objectives is that to precisely specify the semantics of the GFP function that retrieves the values for a frame slot, we must specify the inheritance semantics to be used. However, different FRSs use different inheritance mechanisms [Kar92]. Conformance with a specific semantics for inheritance would require either altering the inheritance mechanism of a given FRS (violating the no-legislation goal), or emulating the desired inheritance mechanism within the implementation of the protocol (violating performance and generality, since the inheritance method used by that FRS is inaccessible through the protocol).

Currently GFP has many LISP dependencies. Connecting GFP to an FRS implemented in some other language should be straightforward using LISP foreign-function calls, but implementing a user-callable GFP implementation in a different language would require significant effort. Also, precision and generality are conflicting goals. If we precisely model various characteristics of the FRSs in the GFP, the protocol will not be general, because the detailed features of one FRS may not be shared by another.

4.3 Proposed Design of the GFP

The most important decision in the design of the GFP was whether the GFP should be the “least-common-denominator” or a “superset” of a class of FRSs. In the least-common-denominator approach, the GFP would contain only those features that are supported by most of the systems. In the superset approach, it would aim to support the union of the features of all FRSs. Not surprisingly, our design is a hybrid of the two approaches described. The core design of the GFP is based on the least-common-denominator approach. It can be parameterized in various ways to capture the peculiarities of a FRS that are not covered in the core model. We, however, did not design the GFP to be a superset of all FRSs, because that would make the protocol cumbersome, complex, and more difficult to use.

Our design of the GFP is based on a comprehensive survey of FRSs. Our survey revealed a large variety of system designs [Kar92]. Some of the differences among these systems were significant while others were superficial. We identified those commonalities that are useful for a broad range of applications. The knowledge model of GFP, as presented here, is based on an axiomatic formalization of classes, relations, and functions for the frame ontology in ONTOLINGUA [Gru93] but extends the ontology to include aspects relevant to operational applications (e.g., kinds of inheritance, facets).

For the rest of this section, we present the knowledge model of the GFP along with possible ways to parameterize it. We also consider how the model was designed to keep the protocol generic.

4.3.1 Representational Primitives

A *frame* is an object with which facts are associated. Each frame has a unique name. Frames are of two kinds: classes and instances. A *class* frame represents a semantically related collection of entities in the world. Each individual entity is represented by an *instance* frame. A frame can be an instance of many classes, which are called its *types*. A class can also be an instance, that is, an instance of a class of classes (a *meta-class*).

Information is associated with a frame via *slots*. A slot is a mapping from a frame and a slot name to a set of values. A slot value can be any LISP object (e.g., symbol, list, number, string). Slots can be viewed as binary relations; GFP does not support the explicit representation of relations of higher arity.

Facets provide information about slots. Some facets pertain to the values of a slot; for example, a facet can be used to specify a constraint on slot values or a method for computing the value of a slot. Other facets may describe properties of the slot itself, such as documentation. In GFP, facets are identified by a facet name, a slot name, and a frame. A facet has as its values a set of data objects.

A *knowledge base*, or KB, is a collection of frames and their associated slots and values. Multiple KBs may be in use simultaneously within an application, possibly serviced by different FRSs. Frames in a given KB can reference frames in another KB, provided that both are serviced by the same FRS.

4.3.2 Inference Mechanisms

Our survey of FRSs revealed that three forms of inferences are most prevalent: *subsumption reasoning*, limited forms of *constraint checking*, and *slot value inheritance*. Consequently, GFP supports only three types of inference.

In GFP, it is possible to specify and query subsumption (or class-subclass) relationships. Some FRSs require subsumption relationships to be specified when frames are created. In contrast, FRSs that perform automatic classification infer the subsumption relationships by comparing class definitions. The GFP operations allow the user to interrogate any class-subclass and class-instance relationships, no matter how the relationships were derived.

GFP recognizes type and number restriction constraints on slot values that are specified as facets. It does not, however, evaluate these constraints.

Inheritance in GFP is based on the use of *template* and *own* slots. A template slot is associated with a class frame and may be inherited by all the instances of that class. An own slot can be associated with a class or instance frame and cannot be inherited. Inheritance in GFP can be characterized as follows. The values of a slot is computed by combining the values of own slots and template values for that slot of all superclasses, provided those values

do not conflict. GFP allows the use of different semantics for “combining” and “conflict,” to support a range of inheritance methods as described in the next section.

4.3.3 Parameterization Using Behaviors

GFP supports extensions to the core knowledge model described earlier to capture the features of FRSs that are not covered by the core model. This diversity is supported through behaviors, which provide explicit models of the FRS properties that may vary. An application program can query the value of behaviors and use the result in executing the code specific to the value of that behavior.

For example, the behavior `:inheritance` can be used to specify the model of inheritance used by an FRS. Two possibilities are currently supported:

override — The presence of any local value in a given slot of a frame blocks inheritance of any values for that slot from superclasses of the frame.¹

incoherence — A slot inherits from its superclasses all values that do not lead to any incoherence in the slot values. We say that the slot values are coherent, if they do not violate any constraint associated with the slot.

Imagine that slot `color` records all colors visible on the surface of an animal, and that the default at class `Elephant` for `color` is `gray`. Suppose that the elephant Clyde has `blue` as a local value if `color`, to reflect the color of Clyde’s eyes. For the **override** inheritance semantics, the user-visible value of the `color` of Clyde would be `blue`, whereas for **incoherence**, the `color` of Clyde would be `{blue,gray}`. In the first case, the local value blocks inheritance of the default value, whereas in the second case, inheritance is not blocked because no constraint specifies that `gray` and `blue` are inconsistent values.

Parameterizing an application based on the values of behaviors, is a better solution than hard coding the peculiarities of an FRS in an application. In practice, we have found that there are some behaviors that are common to some applications but not necessarily to all applications making them impossible to include in the knowledge model. Having a library of such behaviors makes it easier to apply the GFP to a new FRS.

4.3.4 Set of Operations

The GFP defines a programmatic interface of common operations that span the different object types in the knowledge model, namely, *knowledge bases*, *frames*, *classes*, *instances*, *slots*, and *facets*. Three categories of operations are supported for each object type: *retrieval operations*, *manipulator operations*, and *iterators*. Retrieval operations extract information about objects and object values; functional operations retrieve a value; and relational operations test whether a relation holds between an object and some value(s). Manipulator operations create, destroy, and modify objects.

¹This form of inheritance is sometimes referred to as *specificity* inheritance.

As an example, consider operations on slots. The most commonly used retrieval operation on slots is `get-slot-values` which retrieves the values of a slot given a frame name and a slot name. The operation `slot-value-p` tests whether a given value is one of the values of the slots of a given frame. An example of manipulator operation on slots is `create-slot` which is used to define a new slot of a frame. To operate on all the values of a slot, one may use the iterator function `do-slot-values` that can have an arbitrary body which is evaluated by iterating over the slot values.

In addition, GFP supports operations on behaviors. Retrieval operations obtain information about the behaviors supported by GFP in general, the behaviors that a given FRS supports, and the behaviors that are enabled for a particular KB.

4.4 Implementation

Each GFP operations is implemented as a Common LISP Object System (CLOS) method. We identified a subset of GFP operations, called the *kernel* operations which can be used to implement every other operation not in the kernel. We have created a default method for all the nonkernel operations that defines them in terms of kernel operations. For example, the default method for `slot-value-p` calls the kernel operation `get-slot-values`. The kernel consists of roughly 30 operations, whereas the total GFP operations are over 200. The default methods can be overridden to improve efficiency or for better integration with development environments. We can create GFP implementations for new FRSs quickly because only the kernel methods need to be implemented.

GFP back-ends exist for LOOM[Mac91], SIPE-2[Wil90], THEO[MAC⁺89], and ONTOLINGUA[Gru93]. The LOOM and ONTOLINGUA back-ends have been used in conjunction with the GKB-Editor. A read-only back-end exists for CLASSIC. Additional back-ends are planned for Algernon [Cra90] and object-oriented databases.

4.5 Logging Facilities

The GFP implementation also provides a facility to capture, in the form of a log, all KB update operations executed in a user session. The log can be used in a variety of ways. For example, in Section 7, we describe how we use the logging capabilities of GFP to support multiuser access to knowledge bases. The log can also be used for propagating updates between replicated copies of a knowledge base at multiple sites. It is also used by the GKB-Editor to support “undo” for user operations.

4.6 Evaluation of the Protocol

In this section, we discuss difficulties that have arisen while creating GFP back-ends for different FRSs, and we present a quantitative evaluation of GFP.

4.6.1 Qualitative Evaluation

In this section, we discuss incompatibilities between the GFP knowledge model and the models of LOOM and CLASSIC.

The knowledge model of LOOM fits the GFP model fairly closely. For most GFP operations, we were able to identify equivalent LOOM functions. The key difficulties we faced were in capturing the concept-definition language of LOOM and its contexts. Attributes of LOOM classes are specified through complex definition expressions. As GFP does not support concept definition languages per se, we developed a mapping between the LOOM language and GFP facets. For example, the `:at-most` concept construct in the LOOM concept definition language maps to the `:numeric-maximum` facet in GFP. At present, we don't have any notion of contexts in GFP. Currently, each GFP KB maps to a single LOOM context. This representation does not capture the feature that contexts in LOOM can be organized into a hierarchy, and can inherit assertions from the parent contexts.

CLASSIC differs from GFP to a greater extent than LOOM is. First, in CLASSIC, one cannot redefine concepts, whereas one can redefine concepts in the GFP. The ability to redefine concepts allows us to create concept definitions incrementally. For example, our LOOM/GFP implementation exploits the ability to redefine concepts to add facets one at a time. A possible way to mimic the behavior of CLASSIC in the GFP is to maintain a separate structure that collects several updates corresponding to a concept definition and sends them to the classifier only when the concept definition is complete.

Second, in CLASSIC, a slot must be defined before the concept that utilizes it, whereas the GFP assumes the opposite. This makes it difficult to implement the GFP operation `create-slot`, which has two required arguments: slot name and the classes it is attached to. We faced a similar problem while working with LOOM, but in LOOM, either slots can be created before classes or *vice versa*. Therefore, our LOOM/GFP back-end assumed that a class has been defined before its slots, and thus, provides a subset of the functionality supported by LOOM. To address the problem for CLASSIC, we had to revise the `create-slot` operation to accept only the slot name as the required argument.

Third, CLASSIC does not have strong typing. It knows if a slot is valid in a knowledge base, but the slot itself does not know its domain. To capture this in the CLASSIC/GFP back-end, we indicate the applicability of a slot to a class by giving a restriction on the slot. A function such as `get-frame-slots` will then have to check the restriction to compute the correct value for the slots of a class.

Finally, CLASSIC supports open world reasoning by offering an operation for "closing" slots on an instance to indicate that there can be no more values of this slot. Such a behavior can be represented in the GFP by introducing a new facet called `:closed` that can have values T or nil.

In summary, GFP was able to cover a substantial part of the FRS features that we considered. It was not complete in many cases, but we were able to deal with them by minor extensions to our initial design. Therefore, we believe that GFP effectively meets the goal of a generic API for knowledge bases.

4.6.2 Quantitative Evaluation

Our experimental evaluations indicate that the performance penalty for using GFP is acceptable. The overhead incurred by the use of the GFP is largest for fast operations and smallest for slow operations. Using a LOOM implementation of GFP, we compared the running times of key GFP kernel operations with the corresponding LOOM operations. The results showed the GFP operations to be 1% to 50% slower (depending on the operation). The high overhead costs resulted for operations without direct counterparts in LOOM. For example, GFP provides an operation for retrieving a frame when given an identifier but LOOM has no such operation. Instead, it provides separate operations for instances and classes. The GFP operation must consider whether the name corresponds to a class or an instance in order to invoke the appropriate underlying LOOM operation. We note that on an absolute scale, the overhead is very small in this case (approximately 0.02 milliseconds).

For directly comparable operations, the upperbound on overhead was 35%. Much of the increased execution time results from activities common to all GFP operations. Thus, the overhead is high on a percentage basis for fast operations such as slot value retrievals (35% for a 0.3-millisecond operation), but low for more expensive operations such as retrieving all instances of a class (1% for a 16-millisecond operation).

4.7 Related Work

Both GFP and Knowledge Interchange Format (KIF) [GF92] seek to provide a domain-independent medium that supports the portability of knowledge across applications. KIF is more expressive than the GFP as KIF is a comprehensive first-order representation formalism whereas the GFP captures a subset of first-order logic that represents class hierarchies. Ontolingua [Gru93] is a set of tools for writing and analyzing KIF knowledge bases along with translators for mapping KIF KBs to specific FRSs. KIF and ONTOLINGUA are declarative representation languages; GFP is a procedural interface for accessing representation structures. KIF and Ontolingua are designed for use in sharing a large corpus of knowledge at specification time, through the use of translators. GFP is designed for runtime access and modification of existing KBs. GFP is similar to Knowledge and Query Manipulation Language (KQML) [PFPS⁺92] in that it provides a set of operations defining a functional interface for use by application programs. The KQML operations provide a higher level of interface that is oriented toward agent communication. For example, an agent may query an FRS using a KQML “performative.” The KQML allows an agent to express the action of querying, but provides no language to express the query itself. (The query could be expressed using GFP.) Thus, GFP is complementary to KQML.

5 Generic Knowledge Base Editor

The knowledge representation community has long recognized the need for graphical knowledge-base browsing and editing tools to facilitate the development of complex knowledge bases. However, the past approach of developing KB editors that were tightly wedded to a single FRS

is impractical [KC84, LG90]. The substantial efforts required to create such tools become lost if the associated FRS falls into disuse. Since most FRSs share a common core functionality, a more cost-effective approach is to amortize the cost of developing a single FRS interface tool across a number of FRSs. Another benefit of this approach is that it allows a user to access KBs created using a variety of FRSs through a single graphical user interface (GUI), thus simplifying the task of interacting with a new FRS. Finally, most past KB editors have had essentially the same functionality, presumably because each new system must be built from scratch rather than being built on a previous implementation.

The GKB Editor is a generic editor and browser of KBs and ontologies — generic in the sense that it is portable across several FRSs.² This generality is possible because the GKB Editor performs all KB access operations through GFP. To adapt the GKB Editor to a new FRS, we need only create a GFP implementation for that FRS — a task that is considerably simpler than implementing a complete KB editor. The GKB Editor and the Stanford Ontology Editor [FFPR95] have been the most significant applications driving the development of the GFP. They have driven the addition of new operations to GFP, and they have challenged the portability of GFP because the GKB Editor has been used in conjunction with several FRSs.

The GKB Editor contains a number of relatively advanced features, such as incremental browsing of large graphs, KB analysis tools, operation over multiple selections, cut-and-paste operations, and both user and KB-specific profiles. The GKB Editor is in active use in the development of military-application planning KBs and ontologies for LOOM [Mac91] at several sites, for SIPE-2 KBs, and for THEO [MAC⁺89] KBs. It is used daily in the development of EcoCyc, a biological KB containing more than 10,000 frames that is accessed daily via the World-Wide Web by scientists from around the world [KRPPT96].

5.1 Viewing and Editing Knowledge Bases

The GKB Editor offers three different ways to view parts of a KB. The user can view the KB as a class-instance hierarchy graph, as a set of interframe relationships (this is roughly analogous to a conceptual graph representation, a semantic network, or an entity-relationship diagram), or by examining the slot values and facets of an individual frame. A set of editing operations appropriate to each view has been defined so that the displayed objects can be manipulated directly and pictorially.

5.1.1 Class-Instance Hierarchy Viewer

The standard means of viewing a KB is as a class-instance hierarchy graph. Each node in the graph represents a single class or instance frame, and directed edges are drawn from a class to its subclasses and from a class to its instances. Multiple parentage is handled properly.

The hierarchy is normally browsed incrementally. The roots of the hierarchy graph are either computed or specified by the user, and the graph is expanded to a specified depth. If

²Development of the GKB Editor was not supported under this Rome Laboratory project; we summarize the Editor in this report because it is a complementary outgrowth of the work supported by this project.

a particular node has more than a designated number of children, the remaining children are condensed and represented by a single node. Unexpanded nodes are visually distinguished from expanded nodes. The user can browse the hierarchy by clicking on nodes that are to be expanded or compacted.

The hierarchy viewer can also be used to modify the class-instance hierarchy. Operations such as creating, deleting, and renaming frames, and altering superclass-to-subclass links and class-instance links can all be accomplished with a few mouse clicks.

5.1.2 InterFrame Relationships Viewer

It is often useful to visualize slot relationships in a KB rather than parent-child relationships. For example, if frame *B* is a value of slot *X* in frame *A*, then an edge can be drawn from node *A* to node *B*, labeled *X*. If we recognize that slot *X* represents a relationship between frames *A* and *B*, then this kind of graph is analogous to the view of a KB as a conceptual graph (although our displays do not use all the visual conventions of the conceptual graph community) or to a semantic network. Like the hierarchy view, a relationships view is browsed incrementally. The user specifies a set of frames to serve as roots, and optionally a set of slots to follow (by default, all slots are followed), and the graph is expanded to the designated depth and breadth.

5.1.3 Frame Editing Viewer

The frame-editing viewer allows the user to view and edit the detailed contents of an individual frame. The user may select a frame from the hierarchy viewer or the relationships viewer and display it in a frame-editing viewer which presents the contents of a frame as a graph. Each slot name forms the root of a small tree, with its children being the individual slot values as well as slot facets and their values. Inherited items are distinguished visually from local items, and cannot be edited (although they can be overridden where appropriate).

In addition to duplicating, renaming, or deleting the viewed frame, the editing operations available in this viewer permit adding, deleting, replacing, and editing of slot values, facet values, and annotations. Slots themselves may be added, removed, or renamed, when classes are edited.

5.2 Related Work

Many graphical browsers and editors have been built for individual FRSs. We have built on ideas from a number of these systems including KnEd [Eil94], CODE4 [SL95], and HITS [LG90]. Protege-II is a suite of knowledge-acquisition tools [EPG⁺94]. One of its components supports ontology editing; a second component accepts an ontology as input, and produces as output a specification of a forms-based editor for instance-frames within that ontology.

All the preceding tools are restricted to use with a single FRS. Stanford's Ontology Editor [FFPR95] is a browser and editor for ontologies [Gru93]. Currently the Ontology Editor operates only on Ontolingua ontologies, but because it is implemented using GFP, it could

be used to browse and edit KBs for a variety of KR systems. The WWW implementation of the Ontology Editor is both its biggest advantage and its biggest drawback. The advantage is the easy accessibility of the server; its drawbacks result from the many limitations of the HTTP protocol: most information is presented in textual form, rather than graphically; and displays cannot be updated incrementally, as they can in the GKB Editor.

6 Storage System

This section discusses the goals of the storage system, and the architecture we have chosen to satisfy those goals. We present an empirical evaluation of the storage system. We discuss the rationale for our design in some detail, and then present related work.

6.1 Objectives/Motivation

All existing FRSs process their KBs in data structures that exist entirely in memory, forcing users to read the whole KB into memory from disk before its use. To provide persistence, KBs are written to disk files in their entirety. Saving or loading a KB can therefore be an expensive operation, taking time proportional to the size of the KB. An effective cap is placed on the size of a KB by the amount of time that users are willing to wait for save and load operations, with an absolute cap based on the size of virtual memory. The goal of our storage system is to provide a scalable arrangement in which we can selectively load and save frames without having to load and save the whole knowledge base. Our goal is that in a given session, the time spent in loading the frames is proportional to the number of frames *referenced*; and the time spent in saving the frames is proportional to the number of frames *updated*.

Our storage system design submerges a DBMS within an FRS. The FRS and associated application code act as a client that accesses a DBMS server using a DBMS application-program interface (API). The storage system retrieves frames incrementally, on demand, from the DBMS. Because demand fetching of frames from the DBMS is analogous to page faulting in operating systems, we call this process *frame faulting*. The storage system tracks which frames have been modified and transmits those frames back to the DBMS during a KB-save operation. This architecture allows multiple FRS clients to access a shared DBMS server via a network API. Given this basic architecture, other choices must be made: How should FRS information be organized in the DBMS? How many frames should be retrieved from the DBMS in each access? How can we utilize idle periods to speed up the loading time?

Another important question is: Should the storage system be based on an existing DBMS, or on a new DBMS developed specifically for this project? Developing a DBMS is a very large undertaking, so we preferred to use an existing system (preferably commercial) rather than develop our own, particularly given the small size of our research group. Should we employ a relational or an object-oriented DBMS within the storage system? We considered two candidates for the DBMS to be used in our storage system: the ORACLE commercial relational DBMS (RDBMS), and the EXODUS extensible object-oriented DBMS developed at University of Wisconsin, Madison [FZT⁺92]. We implemented separate prototype storage

systems based on EXODUS and on ORACLE, and evaluated them empirically [KPG94, KP95]. We found that the RDBMS is much easier to work with from a practical point of view, because SQL provides a much higher level of interaction than does the extensive C++ programming necessary to interact with Exodus. Our experiments also showed that the difference between the systems in time to retrieve a frame was not substantial [KP95]. Therefore, we decided to use Oracle for future refinements to the storage system.

6.1.1 Design of a DBMS Schema

To store a KB in a DBMS, we must define a mapping from FRS information (classes, instances, slot values, etc.) to information in the DBMS. Question: Should the storage system employ a *domain schema* or a *generic schema*? Using the domain-schema approach, we define a DBMS schema that is specific to a given KB. For an object-oriented DBMS, for example, the object classes we define would correspond to classes in the KB, which in turn correspond to classes in the real-world domain that is being modeled. That is, we would define a mapping from KB classes, instances, and slot values to classes, instances, and attributes in the OODBMS. A similar sort of mapping would be devised to tables in a relational DBMS, although the mapping would be more convoluted because the object-oriented data model is closer to the frame data model than is the relational data model. Because the DBMS schema is designed based on concepts in the domain being modeled, a different schema is required for every KB that we wish to store.

Conversely, the generic-schema approach defines a single generic DBMS schema (that is, one schema for an Object-Oriented Database (OODB) and a conceptually similar schema for a relational DBMS) that can hold any frame-based KB. One way to construct such a versatile schema is to hide all semantics of frames from the DBMS by treating the DBMS as a simple storage server that treats frames as uninterpreted byte vectors. That is since DBMSs cannot capture the expressiveness of the frame data model, we abandon any attempt to capture frame semantics within the DBMS data model. Section 6.1.2 describes a different generic schema.

These two approaches have several advantages and disadvantages. The chief disadvantage of the domain-schema approach is that neither the relational nor the Object-oriented (OO) data models can capture the rich semantics of the frame data model. The relational model does not allow inheritance of properties from one class to another, nor does it allow multi-valued attributes. The OO data model solves those problems, but it does not allow run-time inheritance of default values, nor facets, nor constraints, nor contexts, nor production rules attached to slot values. In contrast, the generic-schema approach performs all processing of frames within the FRS so that no frame semantics are lost.

Another disadvantage of domain schemas is that because a different DBMS schema is required for every KB schema, either the user must derive that schema manually, which will be error-prone and time consuming, or we must supply complex software for performing the mapping automatically. In contrast, one generic mapping exists from any frame KB to the generic schema.

We chose to design a generic relational schema. That schema consists of five core tables. An example is shown in Figure 2.

Frames					
KB ID	Frame Name	Frame Body	Type	Sequence Number	Number of Parents
1	Army	...	class	0	2
1	Armed-Forces	...	class	0	0
1	Ground-Unit	...	class	0	0
1	5th Brigade	...	instance	0	1

Relations		
KB ID	Relation Name	Body
1	Location	...

KB Mapping	
KB Name	KB ID
Forces	1
Supplies	2

Supers		
KB ID	Class Name	Super
1	Armed-Forces	Army
1	Ground-Unit	Army

Instance Classes		
KB ID	Instance Name	Class Name
1	Army	5th Brigade
1	Army	7th Brigade

Figure 2: The relational schema used to store LOOM KBs in an RDBMS, with sample data.

The *Frames* table contains frame bodies. A frame body is a sequence of bytes that provides LOOM with all the information necessary to create the frame. We place concept and instance bodies together in the same table. Because there are occasions when a frame is referenced without its type being known, we defined a type field that identifies the frame as a concept or instance. We record the number of parents of each frame to enable the storage system to perform certain optimizations. A KB identifier is included in each table, to enable multiple KBs to be stored in one ORACLE database. The *KB Mapping* table associates a KB name with its unique identifier.

Frame bodies are encoded as a list of tuples, where each tuple consists of a slot name and a list of slot values, or a slot name and facet name and list of values. Most bodies will be relatively short, but some may be on the order of several thousand bytes. Originally we employed the standard LISP ASCII representation of this s-expression, but we found that LISP readers are much slower than a special-purpose package that we implemented to encode and decode LISP s-expressions to and from a byte vector. A sequence number is included in the frames table to allow a body that exceeds the DBMS maximum column size to be split into multiple tuples.

The tables *Supers* and *Instance Classes* enable reconstruction of the concept and instance hierarchy. The former lists the superclass-subclass relationships between concepts; the latter stores the relationship between instances and their parent concepts. Separate DBMS indices are built to retrieve the subconcepts of a concept, the superconcepts of a concept, the instances of a concept, and the parent concepts of an instance. This information is necessary because for a concept or instance to be defined in LOOM, all parent concepts must already be loaded, or LOOM will not be able to classify the new frame. Thus, we must be able to quickly determine the parents of a given frame so that those parents can be retrieved if necessary.

6.1.2 The Slot-Granularity Generic Schema

The advantage of a domain schema over a generic schema is that the DBMS query-processing engine can operate on KB data on the server side. The server can achieve high efficiency because it can make use of precomputed indices, and because it can process the query locally, without transmitting data across a network. Of course, if the DBMS cannot represent the full semantics of frame data, then it will not be able to process queries with respect to all frame information. For example, imagine that we wish to query all instances of class Aircraft that have a value for the Payload slot that exceeds 5000, but that the values of this slot are inferred using default inheritance, or using backward chaining. If neither defaults nor production rules are implemented at the DBMS server, then it could not properly answer this query.

A compromise is to design a DBMS schema that partially captures frame semantics, and to restrict DBMS query processing to only those frame slots whose semantics can be captured by the DBMS schema — such as slots whose values are not inferred using defaults or production rules. Such a compromise could be implemented using either a domain schema or a generic schema. We chose to extend the generic schema in Figure 2 in a manner that essentially indexes specific KB slots within the DBMS. We defined three additional tables that store slot values, and one table that stores the slot names that should be indexed for a given KB. Each of the three tables contains two columns: one stores a slot name, the second stores a slot value (the three tables are for slot values that are strings, numbers, and long strings, respectively). We separate the types in different tables because retrieval from an index on numbers is much faster than from the one on strings. The long values cannot be indexed but they should be available so that we can perform a sequential search on them whenever there is a query on a slot with character values.

Like all indexing schemes, this approach increases the space requirements of the storage system because slot values are stored as part of the body as well as in the slot-value tables. Our scheme, therefore, trades the generality and storage space for the speed of loading. We chose to use this slot-granularity schema in addition to the frame-granularity schema in Figure 2 because we performed experiments that showed that faulting a frame from the slot-granularity schema is significantly slower than faulting a frame from the frame-granularity schema. Therefore, the storage system queries the frame-granularity tables during demand faulting, and it queries the slot-granularity tables for indexed queries.

What is the optimal distribution of work between the DBMS server and the FRS clients?

Different distributions can have a very significant impact on overall system performance. However, we argue that the three different families of FRS usage identified in Section 2 would benefit from different distribution strategies, i.e., no approach will be optimal for every family. General principles for designing a strategy are: minimize the transfer of data across the network; and maximize utilization of computing resources rather than performing all computations at a centralized server. It makes sense to evaluate database-like complex queries at the server to take advantage of indices built at the server, and because it is more efficient to transmit over the network only that subset of the data that was selected by the query. But this argument breaks down when applied to computations such as expert systems and simulations that are computing-intensive (and could therefore overload the server), and that repeatedly reference large regions of a KB. In these cases the cost of repeatedly requesting, via a network query, values of the same slot of a frame, or values of multiple slots from the same frame, will most likely exceed the cost of retrieving the entire frame once, and caching it on the client side to allow future accesses without the need to access the DBMS server.

6.1.3 Prefetching

We can decrease the overall latency of the storage system by decreasing the number of demand-faulting operations it performs. The number of demand faults will decrease if frames that would have been demand faulted are already in memory at the time the demand fault would have occurred. We can achieve that state of affairs by prefetching frames from the DBMS before they are demanded by the application, assuming that the cost of prefetching is less than the cost of demand fetching. Prefetching can be cheaper than demand fetching for 3 reasons: (a) prefetching can occur when the client is idle, (b) prefetching can use server processing when the client is busy, and (c) prefetching multiple frames at once may result in lower transfer cost per frame.

Our current strategy never discards frames that have been loaded into memory. Under the assumption that all KBs to be created in the near term will fit in virtual memory. Under this assumption (which we plan to discard in future work), if prefetching occurs only during client idle time, it is guaranteed to improve performance since it will eliminate some future demand-fetch operations, at no cost. The assumption that client idle time exists is more reasonable for interactive KB applications than for computing-intensive applications. Prefetching, however, does place an additional load on the DBMS server; a large number of clients prefetching from the same server will ultimately decrease overall system performance.

A prefetching system has to decide which frames should be prefetched. Even if we assume that prefetching has no cost, and that large portions of the KB can be prefetch-ed, it is still better to first prefetch those frames that are likely to be demand-faulted in the near future. The principle of locality suggests that the frames most likely to be referenced in the future will be related to those referenced most recently. (To save bookkeeping, we consider only frames that have been recently fetched.) We consider three types of frame relationships: a frame that fills a slot in a recently fetched frame X , a frame that is a subconcept of X , and a frame that is an instance of X . Since concepts are more likely to be accessed than instances, the subconcepts of X are the first candidates for prefetching. Next, we prefetch the frames

that fill some slot of X . We do not prefetch instances of X , however, because the number of instances of a class can be large and the probability of access to a prefetched instance is low.

6.2 Implementation Issues

We have implemented a single storage system that works in conjunction with both the LOOM and the THEO FRSSs, which are both implemented in LISP. Most of the storage system code is also written in LISP. The storage system interacts with Oracle by calling the Intelligent Database Interface (IDI), developed by Paramax, [MFO90] to communicate with the RDBMS server from LISP using SQL queries that can be transported over a network. We are not employing the full power of the IDI; we use only the module of the IDI that formulates and unpacks SQL queries. The implementation has been thoroughly tested with both Lucid Common LISP and Allegro Common LISP environments. Henceforth all of our comments about LOOM also apply to THEO unless otherwise noted.

6.2.1 Frame Faulting

A frame fault occurs when an application (or LOOM itself) references a frame F that has not been fetched from the DBMS. When faulting a frame into LISP memory, we retrieve its body from the DBMS by issuing one or more SQL queries. We then call standard LOOM functions to add the frame to the LOOM KB.

If F refers to some other frame G , LOOM requires that G must exist in the KB. F might refer to G because G is a parent or a child of F , or because G is referenced in a slot of F . If G is a parent of F , and G is not currently in memory, the storage system generates a frame fault for G to allow proper operation of LOOM. But if G is an instance of F , or if G is referenced in a slot of F , we create a place holder (stub frame) for G . G itself will be faulted at a later time if there is some reference to it by the application. More precisely, stub frames are required in LOOM but not in THEO because LOOM uses LISP pointers to refer to frames, whereas THEO uses LISPSymbols to refer to frames.

6.2.2 Prefetching Process Architecture

Our implementation of the prefetching employs multiple LISP threads to allow asynchronous operation of three tasks: application code, the client side of the prefetcher, and the DBMS server. Fetching and loading a frame into LOOM requires a significant amount of computation on the client machine, for example, for classification.³ To allow the local processing to occur in parallel with processing by the DBMS server, we divide frame fetching into two components: retrieving data from the database, and inserting the frame into the LOOM KB. Most of the time involved in retrieving data from the database is spent on the DBMS server or in communication. Thus, we can perform data retrieval (DR) in parallel with client processing

³In fact, calling the LOOM classifier when entering a faulted frame into LOOM is not strictly necessary since the DBMS already stores the results (parent classes) of previous classifications of that frame. But in practice we have not yet determined how to bypass the LOOM classifier.

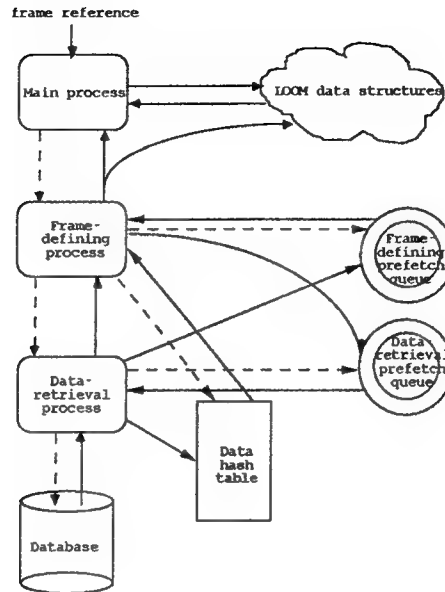


Figure 3: The threads and data structures involved in prefetching. Dashed lines represent requests only. Solid lines represent information flow.

without significantly hurting client performance. The frame-defining (FD) thread is invoked only when a frame is demanded or when the user process is idle. The DR thread can obtain multiple frames with a single query, which does result in an efficiency gain [KP95].

The DR thread runs with the same priority as the application thread (i.e., they time-share). It chooses a frame or set of frames to retrieve (either a demanded frame or frames from a prefetch queue), initiates the appropriate DBMS queries, organizes the resulting frame bodies, and either returns them (if required as part of a demand fetch) or adds the body of each frame to a hash table for storage until needed (if a prefetch operation). The FD thread runs at a lower priority, so it runs only when the other threads block, as in the case of a demand fetch, or are idle.

Figure 3 shows the interaction of the three threads and associated data structures. On a frame fault, the application thread issues a request to the FD thread to create the frame. The FD thread first looks for the body in the hash table, and if unsuccessful, asks the DR thread to query the database. As the frame is being created, any unfetched frames that it references are added to the DR prefetch queue. When the DR thread runs, it checks the DR prefetch queue for frames to prefetch, fetches and adds them to the hash table, and moves the frame references to the FD prefetch queue. When both other threads are idle or blocked, the FD thread checks the FD prefetch queue for frames to define, obtains their bodies from the hash table, and creates the LOOM frames.

When KB size exceeds virtual memory size, it will be important to limit prefetching so that prefetch-ed frames do not displace frames in active use. One approach is to limit the number of related frames that are placed on the prefetch queue. This limitation could be implemented

by adding frames to the prefetch queue with some probability, or through a semantic filter on the slots that are examined for related frames (e.g., to specify on a KB-specific basis those slots from which values are added to the prefetching queue).

6.3 Empirical Results

The goal of the experiments discussed here is to test the scalability of the storage system, that is, the loading time should be proportional to the number of frames referenced and the saving time should be proportional to the number of frames updated. We begin by describing our experimental setup and then discuss our results.

6.3.1 Experimental Setup

The experimental setup for evaluating the storage system consists of test knowledge bases, and software used to conduct the experiments. The test KBs were generated such that their characteristics approximated those of the transportation-planning KB that is driving our work with LOOM [WD94]. All of the test KBs had 100 concepts, all primitive, with just one super each. The concept hierarchy in each knowledge base was the same, regardless of the number of instances. Different test KBs were generated with 500, 1000, 2000, 4000 and 5000 instances. Instances averaged 5 slots apiece, with an average of 2 fillers per slot. Half the slots were filled by integers, with the other half filled by symbols.

Experiments were run using LOOM 2.1, and the February 1993 version of THEO, running on Lucid Common LISP 4.1.1. Both the FRS and the ORACLE server were running on the same workstation, a SPARCstation 10 model 41 with 64 MB of physical memory. LISP was restarted before every trial, to avoid caching effects, and a garbage collection was executed immediately before timing. Overall elapsed times were measured using the LISP `time` function. The time spent in LOOM, THEO, IDI, and the storage subsystem was measured by monitoring key procedures using the monitoring package from Carnegie Mellon University. The CPU time spent in the RDBMS server process was measured using the UNIX `ps` utility to observe total CPU time before and after each experiment. These experiments measured demand fetching times only; the pre-fetcher was disabled.

6.3.2 Experiment 1: Loading Time

The first set of experiments measured the time required to reference some number of randomly chosen instances from knowledge bases of different sizes. Each reference faults in at least one frame from the RDBMS.

Selected results for LOOM and THEO are shown in Figure 4, which breaks the total time spent processing frame faults into several components: the time spent in the RDBMS server, the IDI, our storage system, and the FRS (LOOM and THEO). Figure 4 plots these component times as a function of the number of instances referenced for a fixed KB of 5000 instances. Figure 4(b) shows how the total time for referencing N instances breaks down into time spent in LOOM, our storage system (SSS), IDI, the RDBMS, and other processing (presumably I/O). Figure 4(a) shows an analogous breakdown for THEO.

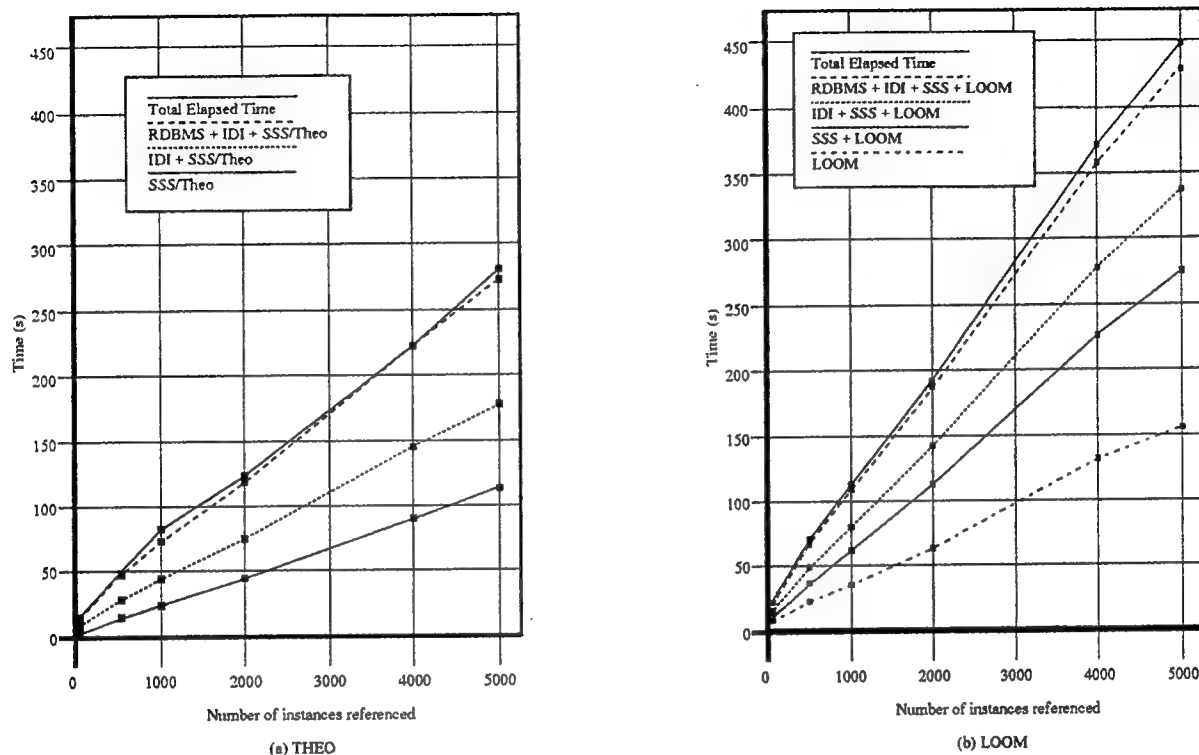


Figure 4: (a) Frame faulting in THEO (top curve) is divided into component times. The bottom curve is time spent unpacking frame bodies and defining them in THEO. The middle curve adds time spent in the IDI (client-side SQL communication); the third curve adds all DBMS server time. (b) Frame faulting in Loom is divided into similar component times.

Figure 5 lets us evaluate the relative merits of loading frames from the RDBMS versus loading from flat files. The relative merits differ for LOOM and THEO because LOOM itself takes significantly longer to load an entire KB of N frames than does THEO. The difference is that LOOM is performing computations (classification) on the KB that THEO is not. Because the same amount of data is transferred for each FRS during incremental loading of N concepts from the same KB, the database costs are about the same. Therefore the ratio of database costs to total costs is higher for THEO than for LOOM. For THEO, loading N instances from the DBMS is eight times slower than loading an entire KB of N instances from a flat file. But for LOOM, loading N instances from the DBMS is only three times slower than loading a KB of that size from a flat file. Therefore the performance of the RDBMS storage system is on a par with that of a flat file when a user references up to 12% of the frames in a KB in a given session for THEO; for LOOM the user can reference up to 30% of the frames for equivalent performance.

In later work, we spent significant efforts tuning the storage system performance. We have decreased the time spent in the SSS, IDI, and RDBMS components in Figure 4. In Allegro

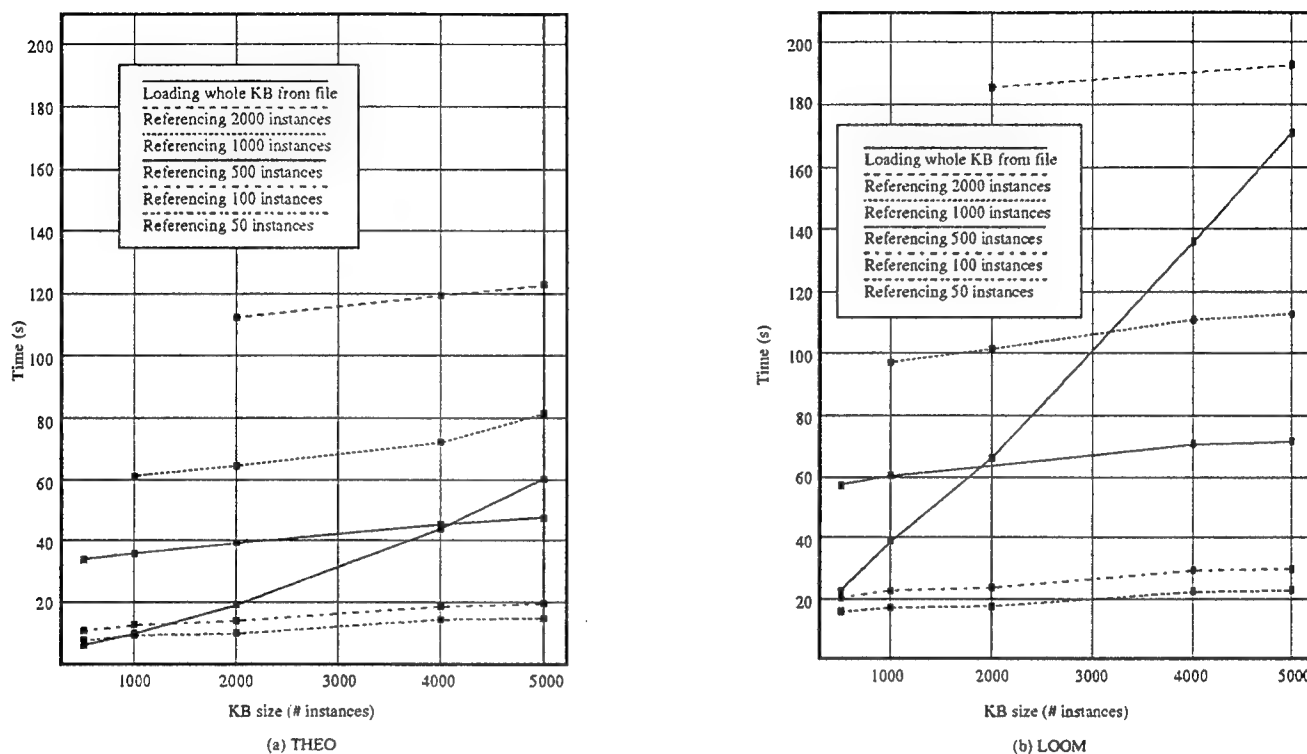


Figure 5: The solid line in each graph shows the time required to load entire knowledge bases of varying sizes from flat files for THEO (a) and Loom (b). The dashed lines show times required to fault in frames from the RDBMS due to references to instances by the application. Each dashed line shows the same number of instance references as a function of KB size. All times refer to total elapsed times. The vertical ordering of dashed lines in each graph and in its legend are the same.

Common LISP, with the latest version of the storage system, the LOOM user can reference up to 69% of the frames in a KB in the same time required to load the flat-file version of the KB.

We take this result to mean that even for THEO the performance of the storage system is acceptable in practice given our assumption that as KB size grows, users will reference only a fraction of its frames in a given session. Note that RDBMS loading also has a different response-time profile than does flat-file loading — flat-file loading requires a long wait at startup time, whereas demand loading hides loading waits across many operations.

6.3.3 Experiment 2: Saving Times

We measured the time required to save updates to some number of randomly chosen instances from KBs of various sizes. To be consistent with traditional LOOM behavior, updates are not written as they occur. Rather, we wait until the user issues a command to save updates,

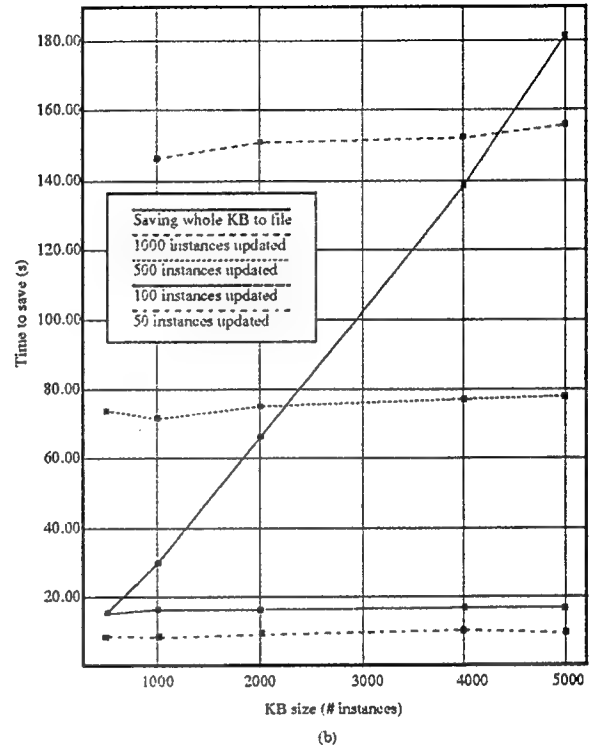
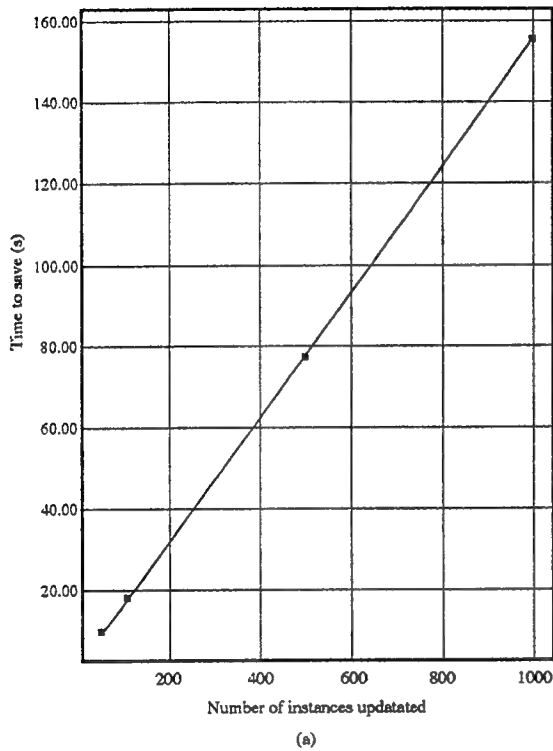


Figure 6: (a) Each data point represents the time to save a given number of instances for a fixed KB of 1000 instances. (b) The solid line shows the time required to save entire KBs of various sizes to Loom flat files. The dashed lines show, for knowledge bases of various sizes, the times required to store a given number of updated instances to the RDBMS. The time required to save 500 updated instances to the RDBMS is about the same as the time required to save an entire KB of 2300 frames to a flat file. All times are total elapsed times.

and then all are written at once in a single transaction. We varied the number of frames updated between 10 and 1000. Selected results are shown in Figure 6. For comparison, we have included the time to save KBs of varying sizes to LOOM flat files (the time is constant for a given KB regardless of the number of frames updated in that KB). KB save times for THEO are similar, and thus are not shown.

Figure 6(a) demonstrates that, as expected, our architecture achieves the goal of saving KB changes in time linearly with the number of updates. Figure 6(b) shows that the time to save frames is not dependent on the size of the KB. Saving N updated frames to the RDBMS is roughly five times slower than saving an entire KB of N frames to a flat file. Therefore, our storage system is faster than the flat file when less than 20% of the KB has been altered.

6.4 Related Work

The Knowledge Engineering Environment Connection couples the KEE FRS with a relational DBMS [AW86] and the IDI couples LOOM with a relational DBMS [MFO90]. In both systems the DBMS and FRS are loosely coupled peers. The advantage of this architecture is that it allows existing information from a database to be imported into an AI environment. Its drawback is that the storage capabilities of LOOM are not enhanced transparently, as in our approach. Users of KEEconnection (and of the IDI) must define mappings between class frames and tables in the RDBMS; KEEconnection creates frame instances from analogously structured tuples stored in the RDBMS, and can store instance frames out to the DBMS. However, only slot values in instance frames can be transferred to the database — class frames cannot be persistently stored using database techniques and cannot be accessed by multiple users. Our approach allows *all* information in a LOOM KB to be permanently stored in the DBMS.

Groups at IBM and at MCC have coupled FRSs to object-oriented DBMSs [MLDW91, BCG⁺88]. The IBM effort differs from our approach in that a KB is read from the DBMS in its entirety when it is opened by a K-REP user, which we believe will be unacceptably slow for large KBs.

None of these researchers have published experimental investigations of alternative implementations, as we are doing. Without systematic experiments, it is impossible to evaluate the relative merits of their architectures.

Markowitz and Chen have developed a system called, *OPM* that implements an object-oriented data model on top of a relational DBMS [CM95]. They use the domain-schema approach, and they have developed automated methods for mapping an OPM object-oriented schema into a relational schema, and for mapping queries in the OPM query language into queries to the underlying relational schema.

7 Collaboration Subsystem

7.1 Motivation/Objectives

The development of large knowledge bases and shared ontologies requires collaboration of multiple people who must simultaneously make contributions. Most knowledge-base development projects involve collaborative development by at most tens of developers. These developers are sometimes geographically distributed, but they all have access to a common high-speed network. The developers perform long sessions of KB-modification operations. A single session sometimes lasts for a number of days (perhaps a week), involving work periods of several hours each day. Developers sometimes save their updates locally after each period before committing any of the updates created over the session to modify the shared KB. These long sessions involve the creation of tens or hundreds of instance frames that are scattered throughout the KB. They may involve modifications to hundreds or thousands of existing frames, and they may also involve schema changes (i.e., creation or modification of class frames). There are also some much shorter work sessions that are similar to traditional

database transactions, although they are small in number.

Most existing FRSs, however, are single-user systems that allow only one person at a time to access a KB, and therefore, are inadequate to support collaborative work. It is not viable or desirable to maintain multiple copies of the knowledge base for each of its users, nor feasible to restrict access to the knowledge base to one user at a time. Using an existing commercial DBMS is an unacceptable alternative for the following reasons. The commercial DBMSs control the operations of multiple users by isolating the database in a way that each user gets the illusion that she/he is the sole user. Such a model works very well for short online transactions, but in a collaborative work scenario described above, it prevents users from working together. The problem becomes especially severe when the users lock large portions of the knowledge base. Instead, what is required is a facility that makes users aware of each other and helps them to work together instead of working in an isolated fashion.

7.2 Proposed Solution

To deal with the problem described above, we allow users to make independent changes to the knowledge base, and when they are done, they merge their changes with the knowledge base. Only those changes of the users are merged with the public copy of the knowledge base that ensure that the overall execution is *serializable* [Pap86].

Serializability can be informally defined as follows. Let a transaction be the set of operations executed by a user. Then an interleaved execution of a set of transactions is serializable if it is equivalent to some serial execution of the same set of transactions in the sense that it leaves the database in the same state and returns the same answers to each one of its users.

Our solution is similar to an optimistic concurrency control technique in which the users make independent changes to the database, but one (or more) of the users making a conflicting change must abort the changes and restart [BHG87]. In our framework, instead of just issuing an abort, we go a step further and assist the users in identifying and resolving their conflicting updates. Furthermore, we use locking only while the conflict-free updates are being deposited into the knowledge base.

Our work so far has considered only those operations that do not involve any schema changes. We plan to consider schema changes in our future work.

For the rest of the section, we describe our solution in more detail. We first describe the model of collaboration that we use. Then we discuss how we record the updates made by a user and the flexible notions of conflict that we have developed. Finally, we describe our strategy for merging the updates of different users.

7.2.1 Model of Collaboration

We maintain a *public* copy of the knowledge base that is readable by multiple KB developers, but cannot be modified directly. All updates to a KB occur in *private workspaces* (or simply, *workspaces*). A developer who wishes to modify an existing public KB creates a private workspace as a child of that public KB; by default the workspace contains all of the information present in its parent and is not necessarily obtained by copying the parent KB. The developer

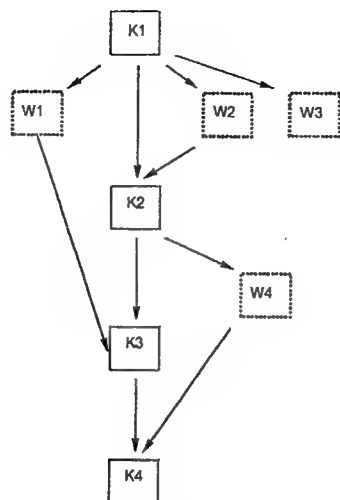


Figure 7: A sample set of relationships among public KBs (K1-K4), private workspaces (W1-W4).

then modifies the workspace by creating new frames, and updating or deleting existing class and instance frames. Those updates do not affect the public KB, and a private workspace can be accessed by only a single user at a time. When the developer has brought the workspace to a satisfactory state, she then *merges* the workspace with the newest state of the public KB. The merge operation detects and resolves conflicts (inconsistencies) between the updates made in the private workspace and updates from which the newest state of the public KB may be derived.

Figure 7 illustrates these concepts more clearly. The boxes labeled K1-K4 represent successive states of a public KB. The boxes labeled W1-W4 represent private workspaces. Each workspace has a single parent that is a public KB. Each state of a public KB either is the initial state, or is derived from a merge of the previous state of that public KB with a private workspace. For example, K2 is derived from a merge of K1 with W2.

Developers may choose not to merge certain private workspaces into public KBs; such workspaces are canceled, in which case all the updates they contain are lost. A workspace is active as long as it has not been canceled, and as long as it has not been merged with a public KB. For example, once K3 has been created by the successive merging of W2 and W1, and if W3 is canceled, then K1 no longer has active child workspaces and it can be canceled.

Because K1 is the parent of W2, and is also the public KB with which W2 is merged, no conflicts can occur during that merge. However, when W1 is merged with K2 to create K3, conflicts may occur between the modifications made in W2 and the modifications made in W1. Developers were updating these two private workspaces simultaneously, and no attempt was made by the KBMS to enforce any consistency with respect to those updates. Consistency is enforced at merge time. Merging will require detections and resolution of conflicts (which we term *arbitration*), and creation of a new state of the public KB.

The users always work with the state of the public knowledge base with which they started.

For example, in Figure 7, even after knowledge base is in state K2, the user of workspace W1 does not see any changes and continues to work with K1.

7.2.2 Recording User Updates

We assume a knowledge base that uses the knowledge model of GFP; that is, it consists of classes, instances, slots, facets and values. The user interacts with the knowledge base by means of GFP operations. A transaction is a set of changes made by a user in a workspace. A transaction is specified as a series of GFP operations. We chose to specify the transactions using GFP operations because then our software can be used with systems that provide GFP back-ends, thus making our approach generic.

With the public copy of the knowledge base, we also record a log of all the changes that have occurred since a given time in the past. Such a log is also called *net-log*. When the updates of a transaction are merged with the public copy, conflicts are detected by comparing the operations in the transaction with the operations in the net-log. For detecting conflicts, only those net-log operations need to be considered that occurred since the user started the session. The net-log is also used for reconstructing previous states of the knowledge base. For example, in Figure 7, if the user of work space W1 requests a value that has been updated in K2, we could reconstruct the old value by using the operations on that value that appear in the net-log.

A log consists of a sequence of records. Each log record is a list with two elements. The first element is the GFP operation itself. The second element is the list of any values that are being overwritten by the current operation; it has a non-null value only when some value is being overwritten and the old value is not available as part of the GFP operation itself.

7.2.3 Implementation of Knowledge Base States

One consequence of not physically copying the complete reference KB when a workspace is created is that it must always be possible to access the reference KB as long as a workspace is being used. Because users can be concurrently modifying the KB, new states of the public KB may be created at any time. We reject the approach of making a complete copy of the current KB when a new state of the KB is created. Thus, the KBMS will have to store the information necessary to recreate the contents of previous KB states until they are no longer needed. This topic will be discussed further in Section 7.2.6.

A state of the KB will have to be accessible as long as some workspace needs to refer to it. Because we do not want to create complete copies of the KB, we will adopt an approach that records only the differences between two KB states. One technique, called *positive* deltas, records the changes necessary to transform an old KB state into a new state. Another technique, called *negative* deltas, records the changes necessary to transform a KB state into a previous state [KC87]. Because an old KB state can be purged when it is no longer needed, we believe that the technique of negative deltas is the correct one to use — positive deltas would require much more work when old KB slots are purged. To resolve a reference to a frame in an old state of the KB, the KBMS starts at the current KB and follows a chain of negative

deltas for that frame (if any) until the state of the frame in that old KB is reconstructed. We will consider whether an intermediate KB state should be removed to improve performance when it is no longer needed by merging its deltas with neighboring deltas, or whether the old KB states should be removed only when no older state is needed.

Let us first describe our model for maintaining previous values of entities. In our current framework, when the user requests for a value from the knowledge base, a value that corresponds to the state in which the user started his or her work is returned. For example, consider a knowledge base state KB_1 that contains a frame `person` with a slot `friend` and an instance `John` whose slot `friend` is initially empty. Suppose two users update the knowledge base, the first one asserting that `Peter` is a friend of `John`, and the second one asserting that `Adam` is a friend of `John`. Let the resulting knowledge base states be KB_2 and KB_3 respectively. If there is another concurrent user who started when the knowledge base was in state KB_2 , and queries for the friends of `John`, she should get `Peter` as an answer even if the query is executed when the public knowledge base is in state KB_3 . There are two approaches to implement such behavior: *delta* and *interval*. Let us briefly consider these approaches and analyze their relative merits.

In the *delta* approach, we keep one copy of the knowledge base and using the log entries compute the desired state of the knowledge base for answering a query. The delta approach can be implemented in two ways: positive and negative deltas. In the *positive delta* approach, we maintain the oldest copy and apply the relevant log records to knowledge base to compute a later state with respect to which the query is to be answered. In the *negative delta* approach, we maintain the newest copy and negatively apply the updates to answer the queries with respect to the older states of the knowledge base. If we apply the negative delta approach to the example considered in the previous paragraph, we store KB_3 , and compute the answer (giving `Peter` and `Adam` as friends of `John`), and negatively apply the entries in the log that are between the knowledge base states KB_2 and KB_3 (which in this case is an operation adding `Adam` as a friend of `John`) to obtain the desired answer (`Peter` is a friend of `John`). In general, more users are interested in the recent states of the knowledge base, and therefore, the negative delta approach is likely to be more efficient than the positive delta approach.

In the *interval* approach, we associate an interval with each entity in the knowledge base indicating the duration for which it exists in the knowledge base. For example, if the value of a slot `salary` is 20K in 1993 and 25K in 1994, the knowledge base contains both of these values. We associate the interval 1 January 1993 to 31 December 1993 with the first value and the interval 1 January 1994 to 31 December 1994 with the second value of the salary. To evaluate any queries on the slot `salary`, we can use a temporal join index [ST95]. While evaluating the query, only the answers that are valid for the duration specified in the query are returned. In the example considered in the previous paragraph, we would get only `Peter` as an answer, because that is the only value valid for the knowledge base state KB_2 .

Over a period of time, the delta approach accumulates a log and the interval approach accumulates past values of entities. Therefore, both approaches require a purging process to remove the information that is no longer necessary. In the negative delta approach, one needs to periodically purge the net-log, and in the interval approach, one needs to periodically

purge the old versions of entities. Clearly, the interval approach requires more storage space than the delta approach, especially because even the entities that have been deleted need to be stored. The delta approach has higher run-time cost because the desired answer must be computed by evaluating the relevant log records. If we assume that the purging overhead is comparable for the two approaches, we are faced with the classical computing vs storage tradeoff. A more detailed evaluation of the two approaches is planned for future work.

7.2.4 Definitions of Conflict

We say that two operations o_1 and o_2 are *conflict-free* if, for any knowledge base state, executing o_1 followed by o_2 leaves the knowledge base in the same final state and returns the same values for each as executing o_2 followed by o_1 . For example, inserting two frames (or nodes) with different names is conflict-free. But if a user wants to change a slot-value (or replace a node) from 2 to 5 and another user wants to change that same value from 2 to 10 then their operations are not conflict-free.

It is useful to consider alternative (and perhaps weaker) notions of freedom from conflict. We say that operations o_1 and o_2 are *partially conflict-free* if, for any knowledge base state, executing o_1 followed by o_2 leaves the knowledge base in the same final state but may not necessarily return the same values for each one of them as executing o_2 followed by o_1 . For example, if two users were trying to create a frame with the same name, then their operations leave the knowledge base in the same state but the user who executes the operation first is successful and the second user would not be able to write anything. We call such operations *partially conflict-free*. Under many situations, it is acceptable to have partially conflict-free operations.

We say that two operations o_1 and o_2 are *conflict-free after modification* if, for any knowledge base state, executing o_1 followed by o_2 leaves the knowledge base in the same final state although it may not necessarily return the same values for each one of them as executing modified o_2 followed by modified o_1 . Modification of operations could be done by some pre-defined functions. For example, suppose one user wants to change a slot name from "height" to "length", while a second user wants to change height from 1.5 to 1.6. If the value of the slot is changed first, we end up with a knowledge base in which length has a value of 1.6. If we change the name first, we cannot change the value of height as that name does not exist any more. In such a situation, if we modify the second operation to mean that we should change length from 1.5 to 1.6, the knowledge base will still be in the same state.

We plan to undertake a more formal analysis of conflict-free after modification in our future work. Specifically, we need to characterize the class of modifications that are acceptable. Also, we need to define the meaning of serializability when the set of operations in the concurrent execution is not necessarily the same as the operations executed by the user.

Sometimes integrity constraint information can be useful in deciding whether operations are conflict free. If the knowledge base contains a constraint that a person may have only one father, two operations that add different value of father to the KB will conflict, but operations adding the same value of father will not conflict (partially conflict-free by the definition above). On the other hand, there may not be such a constraint on the children slot,

and the operations of two users that insert names of two different children are conflict-free.

7.2.5 Conflict Detection

To check the conflicts between a user transaction and the net-log, we must check for each operation in the transaction, if the net-log contains an operation that performs a conflicting update. Each update GFP operation can, in general, involve multiple updates. For example, the `put-slot-values` operation deletes the old slot value(s) and inserts several new slot values. Furthermore, the number of possible GFP operations is quite large (over 200), so that analyzing conflicts between the GFP operations can be quite cumbersome. Therefore, before analyzing the conflicts, we translate the operations into a canonical set of operations that consists of just three operations: `INSERT`, `DELETE` and `REPLACE` on the nodes and edges of the underlying knowledge base graph. Since the number of operations in the canonical set is considerably smaller than the number of GFP operations, the conflict analysis is substantially simplified. Let us now explain our approach in more detail.

We view the knowledge base as a directed graph. The nodes can be classes, instances, slots or values. The edges represent class-subclass (or `isA`), class-instance (or `instanceOf`), class-slot, instance-slot and slot-value relationships. Thus, there are four types of nodes and five types of edges. We use the generic term *entity* to denote either a node or an edge.

For example, in Figure 8, we show a knowledge base KB_1 in a directed graph form. `Employee` and `Person` are classes and represented as nodes. The subclass relationship between them is represented by an edge. `O1` is an instance of `Employee`. `Person` has one slot called `Name`. It is inherited by `Employee`, which has two local slots — `Manager` and `Salary`. The slot values for the instance `O1` are shown as nodes with edges from the slot nodes for `O1`. Since a slot can appear several times in the graph, to identify a slot node uniquely, we must associate it with the frame node it is attached to. For example, the slot `Salary` of the frame `O1` could be identified as `(O1, Salary)`.

There are three types of updates: insert, delete and replace. Replace means modifying a value. A replace operation for edges is not defined. An insert operation on a node is always accompanied by an edge insertion. For example, in the knowledge base KB_1 , if for frame `O1`, we add the value of 20000 for the slot `Salary`, we are inserting a value node 20000 and a slot-value edge from the node `(O1, Salary)`. Similarly, a node deletion is always accompanied by an edge deletion. If we were to delete Adam's `Salary`, we would delete the node associated with the value and at the same time delete the edge between the `(O1, Salary)` slot and the value node. We are interested only in sensible operations. Under an assumption of no duplicates, sensible operations can be defined to mean that one can delete (insert) an entity only if it exists (does not exist) in the knowledge base. In addition, we assume that a node can be deleted only if it has no incident edges and an edge operation makes sense only if the end points of the edge exist in the knowledge base. The set of all operations is the cross product of the set of entity types and the set of operations. With the above model, the conflict analysis between GFP operations reduces to conflict analysis between graph operations.

We assume that every update to the knowledge base is explicitly represented by an operation. For example, if the addition of a default value to a class leads to addition of that value

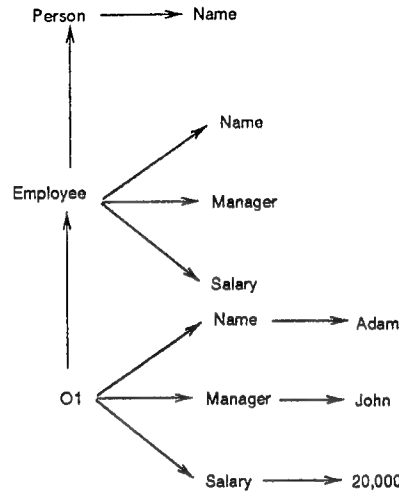


Figure 8: The knowledge base KB_1

to each each of its instances, we represent the addition by a series of operations that add the value to the class and each of its instances.

When the operations involve distinct nodes or edges, they are trivially conflict-free. Therefore, we consider only the situations when the operations involve the same entity.

Table 1 shows the conflict matrix for operations such that both of them either operate on a node or both operate on an edge. To explain the table, and to keep the discussion concrete, consider the case of slot operations. For two slot operations to conflict, they must refer to the same frame and involve the same slot; that is, if the operation is on a slot value X , its frame name and the slot name have to be the same in the two operations under consideration. Consequently, while showing the operations in Table 1, we omit the frame name and the slot name of an operation.

When both operations attempt to insert the same value in a slot, they are partially conflict-free, because only one of them would succeed. If one operation inserts a slot value and another deletes it, the operations could not be both defined, because for a value to be inserted it must not exist, and for it to be deleted it must exist which is a contradiction. Therefore, we do not analyze conflicts for such situations.

The conflicts resulting from an operation renaming the entity X can lead to three kinds of situations: two operations trying to replace X with another but same value, or two operations renaming X to a different value, or an operation renaming another slot value to X . When two operations try to replace a slot value from X to Y , only one of them will succeed, but they will leave the knowledge base in the same state, and therefore, they are partially conflict-free. When one of the operations wishes to replace X with Y and another X with Z , they conflict, because they have different effects on the knowledge base. Finally, when one operation replaces X with Y and another operation replaces Z with X cannot occur in practice, because for the former operation to happen, X must exist in the knowledge base but for the latter operation, X must not exist, which is a contradiction. However, the operation renaming Y to X would

	Insert(X)	Delete(X)	Replace(X,Y)	Replace(X,Z)	Replace (Y,X) Replace (Z,X)
Insert(X)	P	*	*	*	N
Delete(X)	*	P	N	N	*
Replace(X,Y)	*	N	P	N	*

P – Partially conflict-free N – Not conflict-free
 – operations not defined

Table 1: Conflict Matrix for node operations or edge operations

conflict with the operation that inserts X, because executing them in a different order will have different effects on the knowledge base. An operation that deletes X and another operation that replaces X with Y conflict because one of them intends to retain the item in the KB and the other intends to remove it.

Let us now see how Table 1 is applicable for class operations. Two operations inserting a class into the knowledge base are partially conflict-free. Deleting a class could result in the deletion of several instances, which could potentially conflict with other operations on instances. In our analysis, we assume that every operation is explicitly represented in the log, and therefore, if some instances were to be deleted as a result of class deletion, the log would contain explicit delete operations for those instances and would lead to conflicts with other delete operations. Renaming a class would have implications similar to modifying a slot value. For example, two operations renaming a class with different names conflict, but renaming to the same name are partially conflict-free. Table 1 also generalizes to edge operations. To specify an edge operation, however, we do need to specify both the end points and two edge operations would conflict only if they involve the same end points. Thus, two Insert(X,Y) operations would be partially conflict-free. Since the replace operation is not defined for edges, only the first two columns and first two rows of Table 1 are relevant to edge operations.

7.2.6 Merging User Logs

The merge process proceeds in the following steps: log translation, log simplification, log modification, conflict detection, conflict resolution and log concatenation. Log translation transforms the user log, which is originally represented as a sequence of GFP operations, into a sequence of operations on the underlying knowledge base graph. Log simplification takes a translated log, represented as a sequence of graph operations, and obtains the smallest possible log that has the same effect on the knowledge base as the original log. The simplified log captures the net effect of the changes made by a user in a session. Log simplification is necessary, because while detecting conflicts in a later stage of merging, we do not want to consider any redundant operations. Log modification takes the simplified user log, and

modifies the names of the entities that have been renamed since the time the user started. Thus, log modification ensures that only the current names of entities are used in the user log. Conflict detection compares the user log with the net-log and identifies conflicting operations. Conflict resolution resolves the conflicts identified in the conflict-detection phase. In some cases, the conflict resolution step may include suggestions to the user on possible ways to resolve conflicts. Once the user log and net-log are free from any conflicting operations, the user log can be simply concatenated to the net-log to obtain the new net-log. Since the user log and net-log are conflict-free, the resulting net-log is guaranteed to be serializable. Let us now consider these steps in more detail.

Log Translation

Typically, each GFP operation would translate into a series of operations on the underlying knowledge base graph. As an example of this translation, we consider a few GFP operations here. For example, the `put-slot-value` operation would translate to the deletion of all the old value nodes and insertion of a new value node. As a more involved example, consider `create-class` operation. Usually, the parameters of this operation would also include the names of superclasses and slots. If there was only one superclass and no slots specified, then the operation would be equivalent to an `insert-node` operation, which also inserts a node and an edge. In addition, the class would inherit all the slots (except for the local slots) from the superclass, which would translate to inserting an edge corresponding to each slot. If some of the inherited slots had default values, there would be node insertions corresponding to the default values. If more than one superclass was specified, then each superclass would correspond to an `insert-edge` operation between the superclass and the class being inserted. In addition, each slot would translate to a `node-insert` operation.

Log Simplification

Log simplification uses a collection of simplification rules. For example, the insertion of an entity followed by its deletion can be simplified to a null operation. Similarly, if an operation inserts an entity A, and a following operation replaces A with B, the two operations can be simplified to an operation that inserts B. In general, if there are two successive operations on the same slot value can always be simplified to one operation. Therefore, in the simplified log, there is only one operation on each entity.

The simplification algorithm proceeds by a linear scan of the log. For each operation in the log, we check whether there was a previous operation on the entities appearing in the operation. We maintain a table of previous operations that records the previous operation executed on each entity. If there was no previous operation on the entities, we add the operation to the simplified log and to the table of previous operations. If there was a previous operation on the entities, we simplify the two operations and replace the previous operation by the simplified operation — both in the simplified log and in the table of previous operations. The complexity of the simplification process grows linearly with the size of log.

Log simplification is also necessary that we have only one operation on each entity and

detect conflicts between the net effects of operations. For example, if net-log contains two operations, one renaming A to B and the other renaming B to C, the simplification ensures that the net-log contains only the operation renaming A to C. If any conflicts are detected between the simplified operation and an operation in the user-log, we inform the user of the conflicting operations and the person(s) who executed them. Therefore, with each simplified operation in the simplified net-log, we also record the operations that lead to the simplified form. In addition, with each operation, we record the user ID of the person who executed it.

Log Modification

The first task of log modification is to modify the operations in the user-log so that they use the most recent names of entities. Modification entails scanning the user-log, and for each entity in the net-log, checking whether it has been replaced in the net-log. If yes, the operation in the user-log is modified to use the new name of the entity. Suppose the net-log contains an operation renaming a frame from A to B, and the user-log contains an operation adding a slot value to A. In such a case, user operation must be modified to add the slot value to frame B. It can be implemented efficiently if the net-log is indexed in a way that we can retrieve the most recent name of any entity. With an index of most recent names, the complexity of such modification is linear in the size of user-log. (In addition, there is the cost of indexing the net-log which is proportional to $n \log n$ where n is the size of net-log.)

The second task of log modification is to change the user-log to take into account that the knowledge base is not in the same state as the state when the user started. For example, suppose the initial value of a slot when the user starts is 20. Suppose the net-log contains an operation that adds a second value 30 to this slot. Let the user-log contain a put-slot-value operation asserting the value of the slot to be 40, which would remove the old value of 20 and add the new value of 40. Since by the time the log is merged, the slot has another value of 30, which conflicts with the put-slot-value operation in the user-log; the value 30 must be deleted, and the user should be informed of this additional effect of her operation. To achieve this, we must encode in the log sufficient information that can be used during modification. We solved this problem by introducing wild cards in the user log. In the above example, in addition to representing in the log the deletion of slot value 20, we introduce an additional delete operation in which the slot value is replaced by a wild card, meaning all the values of that slot should be deleted. The operation containing the wild card serves as a pattern that is matched against all the entities appearing in the INSERT operations in the net-log, and we generate delete operations for matching entities. In our first attempt at implementing this modification, for every pattern in the user-log, we scan the net-log for matching entities. The worst case complexity of this process is proportional to the product of the sizes of the net-log and user-log.

The modification described above required us to revisit the initial formulation of the conflict matrix. (Recall that we represent conflicts between pairs of operations as a matrix.) In the initial formulation, we had ruled out the possibility that the user-log may contain a delete operation on an entity that is being inserted in the net-log (because the user cannot delete an entity which does not exist). But due to the modification considered above, such a

situation can occur in the modified log and is not really a conflict. Taking this into account in the conflict matrix is trivial because the operations which were previously considered impossible are in fact conflict-free.

We perform conflict detection before log modification because the former is unaffected by the latter, but the converse is not true. There are two kinds of operations that can be involved in a conflict: an operation renaming a frame from A to B, which could conflict with another operation renaming A to C; and an operation inserting A which could conflict with an operation renaming B to A. The modification of the operation renaming A to B would modify it to use the current names of A and B. If in the net-log, A has been replaced with C, there is in fact a conflict with the user operation, which should be detected before any modification is done. In the net-log, B could not be replaced by C, because for the user to be able to replace A with B, B must not exist in the public copy when the user started. In the meantime, the only operation on B in the net-log could be the one that inserts it. Thus, it is unnecessary to modify the name of B. Next, consider an operation in the user-log that inserts A. In the net-log, A could not be replaced by B, because for the user to be able to insert A, A must not exist in the public copy when the user started, and in the meantime, the only operation on A in the net-log could be the one that inserts it. Again, no modification of A is necessary. In a similar way, we can argue that no modification of the net-log is necessary before conflict detection.

Log Concatenation

Once conflicts have been detected and the user-log modified as above, it does not contain any operation that conflicts with some operation in the net-log and is ready for concatenation. Log concatenation involves simply appending the user-log to the net-log. If we are given several logs of more than one user we merge them in the order of their ending time stamps. For example, if the log of first user has timestamp t_1 , the log of the second user has time stamp t_2 , and $t_2 < t_1$, we merge the log of the second user before the log of the first user.

Log concatenation always produces serializable executions because the concatenated transaction does not contain any operations that conflict with the ones in the net-log.

Over a period of time, the net-log will grow in size. To keep its size manageable, we would like to purge the sessions that are no longer necessary. The operations of a user T1 can be purged from the net-log if all the users who started their session before T1 have committed their changes.

Since the net-log can be large, we store it in the Oracle DBMS. The net-log is stored in two tables: the transaction table and the log table. (A transaction is defined as the set of GFP operations executed by a user in one session.) The fields in the transaction table are knowledge base ID, transaction ID, user ID, number of operations in the transaction, the time when transaction began and the time at which the updates made by the transaction were merged with the public copy of the knowledge base. The transaction IDs are unique for a given knowledge base and are system generated. We store in Oracle those logs whose updates have been successfully applied to the knowledge base. The transaction table is indexed on the knowledge base ID, transaction ID and merge timestamp.

Each entry in the log table stores a log record of some transaction. (A log record is a 2-tuple consisting of a GFP operation and any values overwritten by that operation.) To distinguish the log records of different transactions, each entry in the log table records knowledge base ID, transaction ID and the sequence number of the log record within the log records of that transaction. Each log table entry also contains the frame name that the operation refers to, and the string representation of that log record. The log table is indexed on the knowledge base ID, transaction ID and frame name.

7.3 Implementation

We have implemented the solution described above. We performed extensive testing of the implementation for merging using some sample logs collected for a knowledge base that is under development at SRI. All the logs were conflict-free, which suggests that most of the times the users will work on disjoint portions of the knowledge base. Log simplification resulted in simplified logs in many cases, confirming its utility in practice. A more detailed evaluation is planned as future work.

7.4 Related Work

Many approaches have been proposed for supporting collaborative design [NZ90, Kai90, CK86, KS90]. They tend to be inappropriate for our proposed KBMS for several reasons. First, they often support an overly elaborate model of version maintenance. Some models allow all old versions to be recovered and restored, as opposed to our model where public KBs with no active child workspaces can be discarded. Second, the systems often support a check-in/check-out paradigm with some variation of lock management and change notification. This approach is overly complex, and is likely to reduce the potential concurrency of KB development, particularly if the check-out operation checks out not only a user-specified frame, but the transitive closure of all frames referenced by that frame — which could result in checking out a large portion of a KB because of the high connectivity among frames in many KBs. Our approach allows greater concurrency in development assuming that a large number of modification conflicts are not likely, and the conflicts that do occur can be resolved by a merge operation.

There has been little research in providing FRSs with the of multiple-user access. CYC-L [LG90] provides the most sophisticated knowledge-sharing capabilities of any FRS (but which are nonetheless insufficient for many needs): A user can copy the virtual memory image of a KB that is stored on a master knowledge server to the user's workstation (a process that is extremely time consuming). When the user changes the KB on his workstation, CYC-L transmits knowledge-base updates to the master server. The server maintains its KB in virtual memory, not in persistent storage, and it has no mechanism for controlling concurrent KB updates.

The work that is closest to what we propose is that by Mays et al. [MLDW91], and by Hall [Hal91]. Mays and his colleagues are adding persistence and share-ability capabilities to the K-REP FRS. They independently developed the notion of relaxing the strict consistency

requirements of traditional databases in favor of merging privately developed workspaces (which they call *versions*) that may contain conflicts. Their merge operation is not described in [MLDW91], but our private communications with Mays have convinced us that their models of workspaces and of merging are significantly different from ours, and that a number of approaches to this new paradigm of KB development should be explored.

Hall has developed a model for collaborative work on design databases that is broadly quite similar to what we propose, but differs in a number of specifics. Hall's model involves a tree of workspaces, where the design database stored at the root of the tree is the most public and least tentative conception of the design. Workspaces further from the root hold tentative, exploratory designs that are private to individual users or small teams of users. Multiple users can simultaneously *check out* the same object, B, in a single workspace, and work concurrently on B. B is cached on each user's workstation, but is not locked. *Change notification* transmits updates made to B by one user to all other users who have checked out B. In order for a user to check B back in to a workspace, all change notifications about B must have been integrated into the state of B cached on that user's workstation.

The central difference between Hall's model and our model is that Hall's model relies on the process of change notification to maintain the consistency of each object, whereas our model relies on the merge operation. In Hall's model, inconsistencies can be detected very quickly since change notifications can be sent as users perform updates. In our model, inconsistencies are detected only when an individual user merges his changes into the public KB.

Another difference is that in Hall's model, change notifications are actually processed by "design tools" — application programs that are external to the database system — and therefore design tools are responsible for detecting and handling conflicting updates. In our model, conflicts are detected and handled by the KBMS itself. The question of which model is better will depend on the application.

Hall's model will be better for supporting very close collaboration because change notification proceeds during concurrent work — Hall's model can provide earlier detection of inconsistencies. But some collaborators may not wish to work so closely, and may not wish to know about tentative updates made by other users (updates that may be undone later). Furthermore, updates are often meaningful only in groups, and it is not clear how change notification would provide coherent grouping of updates, whereas the merge operation does. In Hall's model, each application program must detect and handle inconsistencies. This approach requires more work for the programmer than does our model, where the KBMS detects inconsistencies, but Hall's model allows more application semantics to be applied to the detection of inconsistencies. A limitation of Hall's model is that two users cannot simultaneously check out the same object for update in different workspaces, which retains some of the disadvantages of locking. Generally speaking, our approaches are complementary, and further knowledge of their relative merits must be gained empirically.

Another proposal similar to ours uses "history merging" as a concurrency control mechanism for collaborative applications [WK96]. Their merging algorithm is called *import algorithm* and computes the minimal subset of a history in the same way we do log simplification.

It then looks for conflict-free subsets of the history and tries to merge them. The authors use the conventional notions of conflict and assume a relational model. In contrast, we have proposed flexible notions of conflict and have used an object model of the database that is more current. We have also proposed conflict resolution strategy that can modify the user log to deal with some simple conflicts.

There is an extensive body of work studying concurrency control in partitioned networks that has relevance in our context [DGMS85]. A partitioned network arises in a replicated database scenario when a subset of the sites get disconnected from the rest due to communication failure. In the collaboration model discussed in this paper, the users work independently and are therefore disconnected from each other. Concurrency control strategies for replicated databases are classified along two dimensions. The first dimension concerns the tradeoff between consistency and availability; the two extremes are syntactic and semantic. The second dimension concerns the type of information used in determining correctness; the two extremes are syntactic and semantic.

Interestingly, the solution proposed here assumes special relevance in the context of active databases that provide several of the facilities required to implement it [DHDD95]. For example, some of these systems maintain hypothetical updates, that is, the updates that a user wishes to make but does not yet want to apply to the database. The user can make queries with respect to different database states that might result by applying the hypothetical updates. The user logs in our solution can be viewed as hypothetical updates. Thus, our solution can be easily exploited by active databases.

8 Summary and Conclusions

The research results described in this report advance the state-of-the-art of knowledge base management systems in several ways:

- The Generic Frame Protocol provides a generic API for KBs that constitutes a solid foundation for knowledge sharing and software reuse.
- The storage system transparently supports efficient storage and retrieval of knowledge bases in a DBMS and establishes a foundation for the scalability of knowledge bases.
- The collaboration subsystem permits multiple users to access a shared knowledge base in a cooperative way. It supports flexible notions of conflicts between user operations and support for resolving those conflicts.
- The GKB Editor completes the above environment by providing three different viewer/editor utilities for browsing and modifying large KBs.

The work described in this report has opened several new opportunities for further work. For example, we are planning to extend the knowledge model of the GFP to deal with contexts and constraints. In our current design of the storage system, the storage system never removes from its cache the frames that have been retrieved. Such an approach does not work when the

knowledge base cannot fit in the main memory. In our future work, we plan to explore schemes for flushing frames that have been retrieved. We plan to extend our collaboration system to deal with schema change operations that have not been covered so far. We plan to undertake a more formal analysis of log modification. Specifically, we need to characterize the class of modifications that are acceptable. We need to define the meaning of serializability when the set of operations in the concurrent execution is not necessarily the same as the operations executed by the user. Finally, we are interfacing the GKB Editor with the World-Wide Web to make it more easily accessible on a variety of platforms.

In conclusion, we our results provide essential infrastructure for the development of very large knowledge bases that are already beginning to appear. Furthermore, they provide a significant starting point for the tools that will be required for composing, authoring and reusing ontologies.

References

- [AW86] R. Abarbanel and M. Williams. A relational representation for knowledge bases. Technical report, IntelliCorp, Mountain View, CA, 1986.
- [BCG⁺88] N. Ballou, H.T. Chou, J.F. Garza, W. Kim, C. Petrie, D. Russinoff, D. Steiner, and D. Woelk. Coupling an expert system shell with an object-oriented database system. *Journal of Object-Oriented Programming*, pages 12–21, June–July 1988.
- [BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Welssley Publishing Company, 1987.
- [CK86] H.T. Chou and W. Kim. A unifying framework for version control in a CAD environment. In *Proc. of the Twelfth International Conference on Very Large Data Bases*, pages 336–344, 1986.
- [CM95] I. A. Chen and V. M. Markowitz. An Overview of the Object-Protocol Model (OPM) and the OPM Data Management Tools. *Information Systems*, 20(5):393–418, 1995.
- [Cra90] J.M. Crawford. *Access-Limited Logic — A language for knowledge-representation*. PhD thesis, University of Texas at Austin, 1990. Technical report AI90-141.
- [DGMS85] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in Partitioned Networks. *Computing Surveys*, 17(3):341–370, 1985.
- [DHDD95] M. Doherty, R. Hull, M. Derr, and J. Durand. On Detecting Conflict Between Proposed Updates. In *International Conference on Database Programming Languages*, September 1995.

- [Eil94] E.F. Eilerts. KnEd, an interface for a frame-based knowledge representation system. Master's thesis, University of Texas at Austin, 1994.
- [EPG⁺94] H. Eriksson, A. R. Puerta, J. H. Gennari, T. E. Rothenfluh, S. W. Tu, and M. A. Musen. Custom-Tailored Development Tools for Knowledge-Based Systems. Technical Report KSL-94-67, Stanford University Knowledge Systems Laboratory, Stanford, CA, 1994.
- [FFPR95] A. Farquhar, R. Fikes, W. Pratt, and J. Rice. Collaborative ontology construction for information integration. Technical Report KSL-95-63, Stanford University, Knowledge Systems Laboratory, Stanford, CA, 1995.
- [FZT⁺92] M.J. Franklin, M.J. Zwilling, C. K. Tan, M.J. Carey, and David J. DeWitt. Crash Recovery in Client-Server EXODUS. In *Proceedings of the ACM SIGMOD 1992 Annual Conference*, pages 165–174, San Diego, CA, June 1992.
- [GF92] Michael R. Genesereth and Richard E. Fikes. Knowledge Interchange Format, Version 3.0 Reference Manual. Technical Report Logic-92-1, Computer Science Department, Stanford University, Stanford, CA, 1992.
- [Gru93] T.R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993. URL for Ontolingua is <http://www-ksl.stanford.edu/knowledge-sharing/ontolingua/README.html>.
- [Hal91] K. Hall. *A Framework for Change Management in a Design Database*. PhD thesis, Stanford University Computer Science Department, Stanford, CA, August 1991.
- [Kai90] Gail E. Kaiser. A flexible transaction model for software engineering. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 560–567, February 1990.
- [Kar92] P.D. Karp. The design space of frame knowledge representation systems. Technical Report 520, SRI International AI Center, 1992. URL <ftp://www.ai.sri.com/pub/papers/karp-freview.ps.Z>.
- [KC84] T.P. Kehler and G.D. Clemenson. KEE the knowledge engineering environment for industry. *Systems And Software*, 3(1):212–224, January 1984.
- [KC87] R.H. Katz and E. Chang. Managing change in a computer-aided design database. In *Proceedings of the Thirteenth International Conference on Very Large Data Bases*, pages 455–462, Brighton, England, 1987.
- [KP95] P.D. Karp and S.M. Paley. Knowledge Representation in the Large. In *Proceedings of the 1995 International Joint Conference on Artificial Intelligence*, pages 751–758, 1995.

- [KPG94] P.D. Karp, S.M. Paley, and I. Greenberg. A storage system for scalable knowledge representation. In N. Adam, editor, *Proceedings of the Third International Conference on Information and Knowledge Management*, pages 97-104, New York, NY, 1994. Association for Computing Machinery. Also available as SRI International AI Center technical report 547.
- [KRPPT96] P. Karp, M. Riley, S. Paley, and A. Pellegrini-Toole. EcoCyc: Electronic encyclopedia of *E. coli* genes and metabolism. *Nuc. Acids Res.*, 24(1):32-40, 1996.
- [KS90] Henry F. Korth and Gregory D. Speegle. Long-duration transactions in software design projects. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 568-574, Los Angeles, CA, February 1990.
- [LG90] D.B. Lenat and R.V. Guha. *Building Large Knowledge-Based Systems: Representation and Inference in the CYC Project*. Addison-Wesley, Reading, MA, 1990.
- [MAC+89] T.M. Mitchell, J. Allen, P. Chalasani, J. Cheng, E. Etzioni, M. Ringuette, and J.C. Schlimmer. Theo: A framework for self-improving systems. In *Architectures for Intelligence*. Erlbaum, 1989.
- [Mac91] R. MacGregor. The evolving technology of classification-based knowledge representation systems. In J. Sowa, editor, *Principles of semantic networks*, pages 385-400. Morgan Kaufmann Publishers, Los Altos, CA, 1991.
- [MFO90] D.P. McKay, T.W. Finin, and A. O'Hare. The intelligent database interface: Integrating AI and database systems. In *Proceedings of the 1990 National Conference on Artificial Intelligence*, pages 677-684. Morgan Kaufmann Publishers, 1990.
- [MLDW91] E. Mays, S. Lanka, B. Dionne, and R. Weida. A persistent store for large shared knowledge bases. *IEEE Trans. on Knowledge and Data Eng.*, 3(1):33-41, 1991.
- [NZ90] Marian H. Nodine and Stanley B. Zdonik. Cooperative transaction hierarchies: A transaction model to support design applications. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, pages 83-94, Brisbane, Australia, 1990.
- [Pap86] Christos Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, Rockville, MD, 1986.
- [PFPS+92] Ramesh Patil, Richard E. Fikes, Peter F. Patel-Schneider, Don Mackay, Tim Finin, Thomas Gruber, and Robert Neches. The DARPA Knowledge Sharing Effort: Progress Report. In *The Third International Conference on Principles of Knowledge Representation and Reasoning*, pages 777-788, Boston, MA, 1992.

- [SL95] D. Skuce and T.C. Lethbridge. CODE4: A unified system for managing conceptual knowledge. *International Journal of Human-Computer Studies*, 1995.
- [ST95] A. Shrufi and T. Topaloglou. Query Processing for Knowledge Bases Using Join Indices. In *In the Proceedings of 4th International Conference on Information and Knowledge Management*, Baltimore, MD, November 1995.
- [WD94] D. E. Wilkins and R.V. Desimone. Applying an AI planner to military operations planning. In M. Fox and M. Zweben, editors, *Intelligent Scheduling*, pages 685–709. Morgan Kaufmann Publishers, 1994.
- [Wil90] D.E. Wilkins. Can AI planners solve practical problems? *Computational Intelligence*, 6(4):232–246, November 1990.
- [WK96] Jürgen Wäsch and Wolfgang Klas. History Merging as a Mechanism for Concurrency Control in Cooperative Environments. In *Proceedings of RIDE'96: Interoperability of Non-Traditional Database Systems*, Louisiana, Feb 26-27 1996.

DISTRIBUTION LIST

addresses	number of copies
LOUIS J. HOEBEL RL/C3CA 525 BROOKS RD ROME NY 13441-4505	10
SRI INTERNATIONAL 333 RAVENSWOOD AVE MENLO PARK CA 94025	5
ROME LABORATORY/SUL TECHNICAL LIBRARY 26 ELECTRONIC PKY ROME NY 13441-4514	1
ATTENTION: DTIC-DCC DEFENSE TECHNICAL INFO CENTER 8725 JOHN J. KINGMAN ROAD, STE 0944 FT. BELVOIR, VA 22060-6218	2
ADVANCED RESEARCH PROJECTS AGENCY 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
RELIABILITY ANALYSIS CENTER 201 MILL ST. ROME NY 13440-8200	1
ROME LABORATORY/C3AB 525 BROOKS RD ROME NY 13441-4505	1
ATTN: RAYMOND TADROS GIDEP P.O. BOX 8000 CORONA CA 91718-8000	1

AFIT ACADEMIC LIBRARY/LDPE 1
2950 P STREET
AREA B, BLDG 642
WRIGHT-PATTERSON AFB OH 45433-7765

DL AL HSC/HRG, BLDG. 190 1
2698 G STREET
WRIGHT-PATTERSON AFB OH 45433-7604

US ARMY STRATEGIC DEFENSE COMMAND 1
CSSD-IM-PA
P.O. BOX 1500
HUNTSVILLE AL 35807-3601

COMMANDER, TECHNICAL LIBRARY 1
4747000/C0223
NAVAIRWARCENWPNDIV
1 ADMINISTRATION CIRCLE
CHINA LAKE CA 93555-6001

SPACE & NAVAL WARFARE SYSTEMS 2
COMMAND (PMW 178-1)
2451 CRYSTAL DRIVE
ARLINGTON VA 22245-5200

COMMANDER, SPACE & NAVAL WARFARE 1
SYSTEMS COMMAND (CODE 32)
2451 CRYSTAL DRIVE
ARLINGTON VA 22245-5200

CDR, US ARMY MISSILE COMMAND 2
RSIC, BLDG. 4484
AMSMI-RD-CS-R, DOCS
REDSTONE ARSENAL AL 35898-5241

ADVISORY GROUP ON ELECTRON DEVICES 1
SUITE 500
1745 JEFFERSON DAVIS HIGHWAY
ARLINGTON VA 22202

REPORT COLLECTION, CIC-14
MS P364
LOS ALAMOS NATIONAL LABORATORY
LOS ALAMOS NM 87545

1

AEDC LIBRARY
TECHNICAL REPORTS FILE
100 KINDEL DRIVE, SUITE C211
ARNOLD AFB TN 37389-3211

1

COMMANDER
USAISC
ASHC-IMD-L, BLDG 61801
FT HUACHUCA AZ 85613-5000

1

US DEPT OF TRANSPORTATION LIBRARY
F810A, M-457, RM 930
800 INDEPENDENCE AVE, SW
WASH DC 22591

1

AIR WEATHER SERVICE TECHNICAL
LIBRARY (FL 4414)
859 BUCHANAN STREET
SCOTT AFB IL 62225-5118

1

AFIWC/MSO
102 HALL BLVD, STE 315
SAN ANTONIO TX 78243-7016

1

SOFTWARE ENGINEERING INSTITUTE
CARNEGIE MELLON UNIVERSITY
4500 FIFTH AVENUE
PITTSBURGH PA 15213

1

NSA/CSS
K1
FT MEADE MD 20755-6000

1

OCMAO/WICHITA/SKEP
SUITE 8-34
401 N MARKET STREET
WICHITA KS 67202-2095

1

PHILLIPS LABORATORY 1
PL/TL (LIBRARY)
5 WRIGHT STREET
HANSCOM AFB MA 01731-3004

THE MITRE CORPORATION 1
ATTN: E. LADURE
D460
202 BURLINGTON RD
BEDFORD MA 01732

DUSD(P)/DTSA/DUTD 2
ATTN: PATRICK G. SULLIVAN, JR.
400 ARMY NAVY DRIVE
SUITE 300
ARLINGTON VA 22202

DR JAMES ALLEN 1
COMPUTER SCIENCE DEPT/BLDG RM 732
UNIV OF ROCHESTER
WILSON BLVD
ROCHESTER NY 14627

DR YIGAL ARENS 1
USC-ISI
4676 ADMIRALTY WAY
MARINA DEL RAY CA 90292

DR MARIE A. BIENKOWSKI 1
SRI INTERNATIONAL
333 RAVENSWOOD AVE/EK 337
MENLO PARK CA 94025

DR MARK S. BODDY 1
HONEYWELL SYSTEMS & RSCH CENTER
3660 TECHNOLOGY DRIVE
MINNEAPOLIS MN 55418

DR MARK BURSTEIN 1
BBN SYSTEMS & TECHNOLOGIES
10 MOULTON STREET
CAMBRIDGE MA 02138

DR GREGG COLLINS 1
INST FOR LEARNING SCIENCES
1890 MAPLE AVE
EVANSTON IL 60201

MR. RANDALL J. CALISTRI-YEH 1
ORA CORPORATION
301 DATES DRIVE
ITHACA NY 14850-1313

DR STEPHEN E. CROSS 1
SCHOOL OF COMPUTER SCIENCE
CARNEGIE MELLON UNIVERSITY
PITTSBURGH PA 15213

MS. LAURA DAVIS 1
CODE 5510
NAVY CTR FOR APPLIED RES IN AI
NAVAL RESEARCH LABORATORY
WASH DC 20375-5337

DR THOMAS L. DEAN 1
BROWN UNIVERSITY
DEPT OF COMPUTER SCIENCE
P.O. BOX 1910
PROVIDENCE RI 02912

DR PAUL R. COHEN 1
UNIV OF MASSACHUSETTS
COINS DEPT
LEDERLE GRC
AMHERST MA 01003

DR JON DOYLE 1
LABORATORY FOR COMPUTER SCIENCE
MASS INSTITUTE OF TECHNOLOGY
545 TECHNOLOGY SQUARE
CAMBRIDGE MA 02139

MR. STU DRAPER 1
MITRE
EAGLE CENTER 3, SUITE B
D'FALLON IL 62269

MR. GARY EDWARDS 1
ISX CORPORATION
2000 N 15TH ST, SUITE 1000
ARLINGTON, VA 22201

MR. RUSS FREW 1
GENERAL ELECTRIC
MOORESTOWN CORPORATE CENTER
BLDG ATK 145-2
MOORESTOWN NJ 08057

DR MICHAEL FEHLING 1
STANFORD UNIVERSITY
ENGINEERING ECO SYSTEMS
STANFORD CA 94305

DR KRISTIAN J. HAMMOND 1
UNIV OF CHICAGO
COMPUTER SCIENCE DEPT/RY155
1100 E. 58TH STREET
CHICAGO IL 60637

RICK HAYES-ROTH 1
CIMFLEX-TEKNOLOGY
1810 EMBARCADERO RD
PALO ALTO CA 94303

DR JIM HENDLER 1
UNIV OF MARYLAND
DEPT OF COMPUTER SCIENCE
COLLEGE PARK MD 20742

MR. MORTON A. HIRSCHBERG, DIRECTOR 1
US ARMY RESEARCH LABORATORY
ATTN: AMSRL-CI-CB
ABERDEEN PROVING GROUND MD
21005-5066

MR. MARK A. HOFFMAN 1
ISX CORPORATION
1165 NORTHCHASE PARKWAY
MARIETTA GA 30067

DR RON LAPSEN 1
NAVAL CMD, CONTROL & OCEAN SUR CTR
RESEARCH, DEVELOP, TEST & EVAL DIV
CODE 444
SAN DIEGO CA 92152-5000

MR. RICHARD LOWE (AP-10) 1
SRA CORPORATION
2000 15TH STREET NORTH
ARLINGTON VA 22201

MR. TED C. KPAL 1
BBN SYSTEMS & TECHNOLOGIES
4015 HANCOCK STREET, SUITE 101
SAN DIEGO CA 92110

DR. ALAN MEYROWITZ
NAVAL RESEARCH LABORATORY/CODE 5510
4555 OVERLOOK AVE
WASH DC 20375

1

ALICE MULVEHILL
BBN
10 Moulton Street
Cambridge MA 02238

1

DR DREW McDERMOTT
YALE COMPUTER SCIENCE DEPT
P.O. BOX 2158, YALE STATION
51 PROSPECT STREET
NEW HAVEN CT 06520

1

DR DOUGLAS SMITH
KESTREL INSTITUTE
3260 HILLVIEW AVE
PALO ALTO CA 94304

1

DR. AUSTIN TATE
AI APPLICATIONS INSTITUTE
UNIV OF EDINBURGH
80 SOUTH BRIDGE
EDINBURGH EH1 1HN - SCOTLAND

1

DIRECTOR
DARPA/ITO
3701 N. FAIRFAX DR., 7TH FL
ARLINGTON VA 22209-1714

1

DR STEPHEN F. SMITH
ROBOTICS INSTITUTE/CMU
SCHENLEY PRK
PITTSBURGH PA 15213

1

DR. ABRAHAM WAKSMAN
AFOSR/NM
110 DUNCAN AVE., SUITE B115
BOLLING AFB DC 20331-0001

1

DR JONATHAN P. STILLMAN
GENERAL ELECTRIC CRD
1 RIVER RD, RM K1-5C31A
P. O. BOX 8
SCHENECTADY NY 12345

1

DR EDWARD C.T. WALKER 1
BBN SYSTEMS & TECHNOLOGIES
10 Moulton Street
Cambridge MA 02138

DR BILL SWARTOUT 1
USC/ISI
4676 Admiralty Way
Marina Del Rey CA 90292

DR KATIA SYCARA/THE ROBOTICS INST 1
School of Computer Science
Carnegie Mellon Univ
Doherty Hall RM 3325
Pittsburgh PA 15213

DR. PATRICK WINSTON 1
Mass Institute of Technology
RM NE43-817
545 Technology Square
Cambridge MA 02139

DR JOHN P. SCHILL 1
ARPA/ISO
3701 N Fairfax Drive
Arlington VA 22203-1714

MR. MIKE ROUSE 1
AFSC
7800 Hampton Rd
Norfolk VA 23511-6097

MR. DAVID F. SMITH 1
Rockwell International
444 High Street
Palo Alto CA 94301

JEFF ROTHENBERG 1
Senior Computer Scientist
The Rand Corporation
1700 Main Street
Santa Monica CA 90407-2138

DR MATTHEW L. GINSBERG 5
CIRL, 1269
University of Oregon
Eugene OR 97403

MR IRA GOLOSTEIN
OPEN SW FOUNDATION RESEARCH INST
ONE CAMBRIDGE CENTER
CAMBRIDGE MA 02142

1

MR JEFF GROSSMAN, CO
NCCOSC RDTE DIV 44
5370 SILVERGATE AVE, ROOM 1405
SAN DIEGO CA 92152-5146

1

JAN GUNTHER
ASCENT TECHNOLOGY, INC.
64 SIDNEY ST, SUITE 380
CAMBRIDGE MA 02139

1

DR LYNETTE HIRSCHMAN
MITRE CORPORATION
202 BURLINGTON RD
BEDFORD MA 01730

1

DR ADELE E. HOWE
COMPUTER SCIENCE DEPT
COLORADO STATE UNIVERSITY
FORT COLLINS CO 80523

1

DR LESLIE PACK KAEHLING
COMPUTER SCIENCE DEPT
BROWN UNIVERSITY
PROVIDENCE RI 02912

1

DR SUBBARAO KAMBHAMPATI
DEPT OF COMPUTER SCIENCE
ARIZONA STATE UNIVERSITY
TEMPE AZ 85287-5406

1

DR PRADEEP K. KHOSLA
ARPA/ITC
3701 N. FAIRFAX DR
ARLINGTON VA 22203

1

DR CARLA GOMES
ROME LABORATORY/C3CA
525 BROOKS RD
ROME NY 13441-4505

1

DR MARK T. MAYBURY ASSOCIATE DIRECTOR OF AI CENTER ADVANCED INFO SYSTEMS TECH G041 MITRE CORP, BURLINGTON RD, MS K-329 BEDFORD MA 01730	1
MR DONALD P. MCKAY PARAMAX/UNISYS P O BOX 517 PAOLI PA 19301	1
DR MARTHA E POLLACK DEPT OF COMPUTER SCIENCE UNIVERSITY OF PITTSBURGH PITTSBURGH PA 15260	1
DR EDWINA RISSLAND DEPT OF COMPUTER & INFO SCIENCE UNIVERSITY OF MASSACHUSETTS AMHERST MA 01003	1
DR MANUELA VELOSO CARNEGIE MELLON UNIVERSITY SCHOOL OF COMPUTER SCIENCE PITTSBURGH PA 15213-3891	1
DR DAN WELD DEPT OF COMPUTER SCIENCE & ENG MAIL STOP FR-35 UNIVERSITY OF WASHINGTON SEATTLE WA 98195	1
DR TOM GARVEY ARPA/ISO 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
MR JOHN N. ENTZMINGER, JR. ARPA/DIRO 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
LT COL ANTHONY WAISANEN, PHD COMMAND ANALYSIS GROUP HQ AIR MOBILITY COMMAND 402 SCOTT DRIVE, UNIT 3L3 SCOTT AFB IL 62225-5307	1

DIRECTOR
ARPA/ISD
3701 NORTH FAIRFAX DRIVE
ARLINGTON VA 22203-1714

1

OFFICE OF THE CHIEF OF NAVAL RSCH
ATTN: MR PAUL QUINN
CODE 311
800 N. QUINCY STREET
ARLINGTON VA 22217

1

DR GEORGE FERGUSON
UNIVERSITY OF ROCHESTER
COMPUTER STUDIES BLDG, RM 732
WILSON BLVD
ROCHESTER NY 14627

1

DR STEVE HANKS
DEPT OF COMPUTER SCIENCE & ENG'G
UNIVERSITY OF WASHINGTON
SEATTLE WA 98195

1

DR WILLIAM S. MARK
LOCKHEED PALO ALTO RSCH LAB
DEPT 9620, BLDG 254F
3251 HANOVER ST
PALO ALTO CA 94304-1187

1

DR ADNAN DARWICHE
INFORMATION & DECISION SCIENCES
ROCKWELL INT'L SCIENCE CENTER
1049 CAMINO DOS RIOS
THOUSAND OAKS CA 91360

1

DR JAMES CRAWFORD
CIRL, 1269
UNIVERSITY OF OREGON
EUGENE OR 97403

1

ROBERT J. KRUCHTEN
HQ AMC/SCA
203 W LOSEY ST, SUITE 1016
SCOTT AFB IL 62225-5223

1

MISSION OF ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

- a. Conducts vigorous research, development and test programs in all applicable technologies;
- b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;
- c. Provides a full range of technical support to Air Force Material Command product centers and other Air Force organizations;
- d. Promotes transfer of technology to the private sector;
- e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.